Memoria TFG

Optimización y paralelización de algoritmo de búsqueda aleatoria
para su aplicación en la búsqueda de fuentes MEG

08/06/13

Autor: José Mª Pérez Ramos
Tutor: Antonio García Dopico

Control de versiones

versión	Fecha	Hito
0	25/04/13	Creación documento
1	26/04/13	Modificación del índice
2	27/04/13	Primera redacción (borrador) Introducción y Trabajos previos
3	08/06/13	Fin redacción
4	09/06/13	Retoques finales

Índice

1. Trabajo realizado

Durante el semestre mi trabajo ha consistido en desarrollar y optimizar una herramienta software que implementa el algoritmo Solis-Wets para la búsqueda de fuentes MEG, para ello, a principios de semestre tuvimos varias reuniones con los responsables de la biblioteca GAEDA, tras las que me proporcionaron el código de su biblioteca y los datos necesarios para ejecutar dicho algoritmo.

Obtenida la información correspondiente al citado algoritmo Solis-Wets a través del artículo *Minimization by Random Search Techniques. In Mathematics of Operations Research*, reseñado en la bibliografía del presente trabajo, desarrollé en pseudocódigo una versión preliminar del mismo que utilicé a continuación para el desarrollo de la herramienta.

Una vez diseñada la primera versión de la herramienta, ejecuté tanto *valgrind* como *vtune* para evaluar los puntos menos eficientes de la misma, y, tras valorar los resultados, me dispuse a optimizarla.

Comencé por las optimizaciones algorítmicas y continué probando distintas bibliotecas matemáticas.

En este punto percibí que la herramienta tenía lo que yo interpreté como un comportamiento extraño, hecho que me llevó a descubrir una variable que no tenía suficiente resolución para almacenar su valor, por lo que tuve que introducir otra variación sobre el código, aunque el comportamiento extraño era el correcto.

Continué la optimización, en este caso mediante paralelización, con la que, a pesar de disponer de una máquina con 12 núcleos, no pude lograr más de un *speedup* de 5.06x.

A partir de ese momento inicié la vectorización, tema que desconocía y que no estaba previsto en las tareas planificadas al comienzo del trabajo. No obstante, es un tema interesante y me ha resultado muy gratificante abordarlo. Un inconveniente asociado a esta incidencia en la planificación, es que el tiempo dedicado a la vectorización no pude dedicárselo a CUDA, lenguaje que pienso que es ideal para acometer el diseño de este algoritmo.

También dediqué mucho tiempo intentando hacer funcionar el *toolbox fieldtrip* en MATLAB, tanto en Linux como Windows.

Por último me realicé las ejecuciones del programa para comprobar su comportamiento en función de la solución inicial elegida, ya que siempre nos había llamado la atención el hecho de que fuese aleatoria.

Para concluir, me gustaría destacar que todas las semanas se planificaba una reunión de seguimiento con el Grupo de Optimización y Paralelización de Aplicaciones Científicas en la que el resto de integrantes del grupo y yo exponíamos nuestros avances en los distintos proyectos.

2. Resumen

El problema inverso de la búsqueda de fuentes MEG consiste en la obtención de la distribución de los dipolos de corriente (fuentes) en el interior de la cabeza de un paciente a partir de las mediciones de campo electromagnético obtenidas en la superficie (magnetoencefalograma, MEG).

Para obtener estos datos, en el ámbito científico se utiliza el algoritmo *beamforming*, comúnmente aceptado, cuyos resultados ofrecen un pequeño margen de error debido a la naturaleza del problema.

Esta memoria desarrolla el trabajo realizado para optimizar un algoritmo de búsqueda aleatoria, Solis-Wets, utilizado para investigar la posibilidad de su aplicación en el ámbito científico, en sustitución del anteriormente mencionado, *beamforming*.

También se estudiará la acción de encadenar ambos algoritmos, tomando como datos de entrada del algoritmo Solis-Wets aquellos proporcionados como solución por el algoritmo *beamforming* con objeto de minimizar el error en el que éste incurre.

Esta optimización es necesaria para que la alternativa sea viable debido al tiempo necesario en su ejecución, e incluye el uso de bibliotecas auxiliares, así como la paralelización del código.

Para la evaluación del algoritmo se han medido tanto la velocidad de generación de soluciones como el error de la mejor solución tras un número determinado de soluciones generadas.

Como variables para esta evaluación se han tomado distintos compiladores, distintas soluciones de partida, precisión de los datos, así como el uso de distintas bibliotecas matemáticas disponibles.

3. Introducción

A la hora de intentar reconocer patrones asociados a enfermedades o a los distintos procesos cognitivos, lo ideal sería conocer los mecanismos que determinan la actividad interna del cerebro, pero sólo disponemos de las medidas del campo electromagnético en la superficie gracias a los magnetoencefalogramas (MEG) pues se pretenden evitar actuaciones de alto riesgo para el paciente.

El objetivo de a búsqueda de fuentes MEG consiste en encontrar las fuentes (dipolos de corriente) que producen el campo magnético medido por el magnetoencefalograma.

La búsqueda de fuentes MEG se puede catalogar en dos subproblemas:

- Problema directo:
  Conocidas las fuentes, se calcula el campo magnético que se produce en los sensores.
- Problema inverso:
  Los magnetómetros y gradiómetros del casco miden las variaciones de campo magnético en la superficie de la cabeza de un paciente del que se pretende obtener la distribución de actividad eléctrica en el interior de la cabeza.

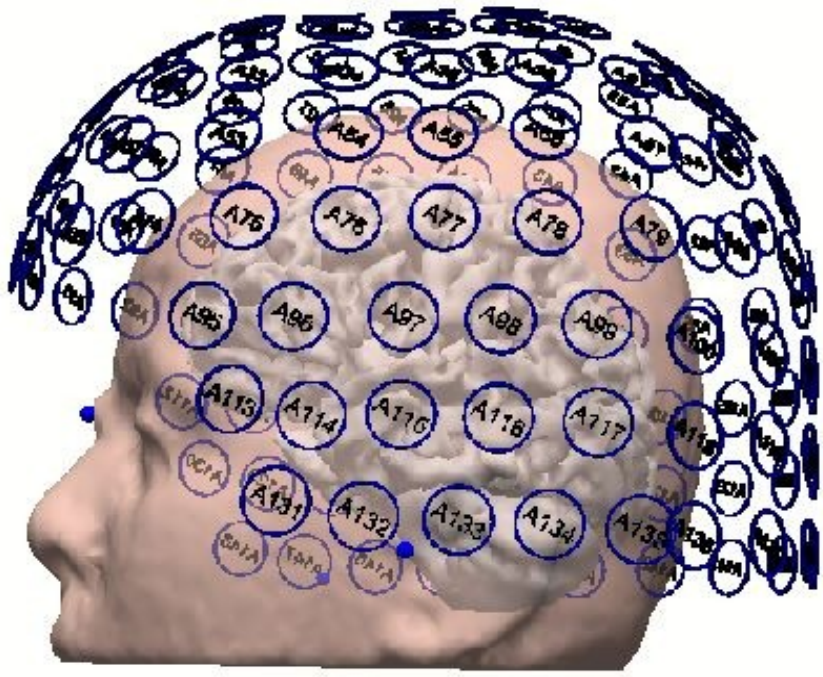

Figura 1: Ejemplo de distribución de los sensores por el casco.

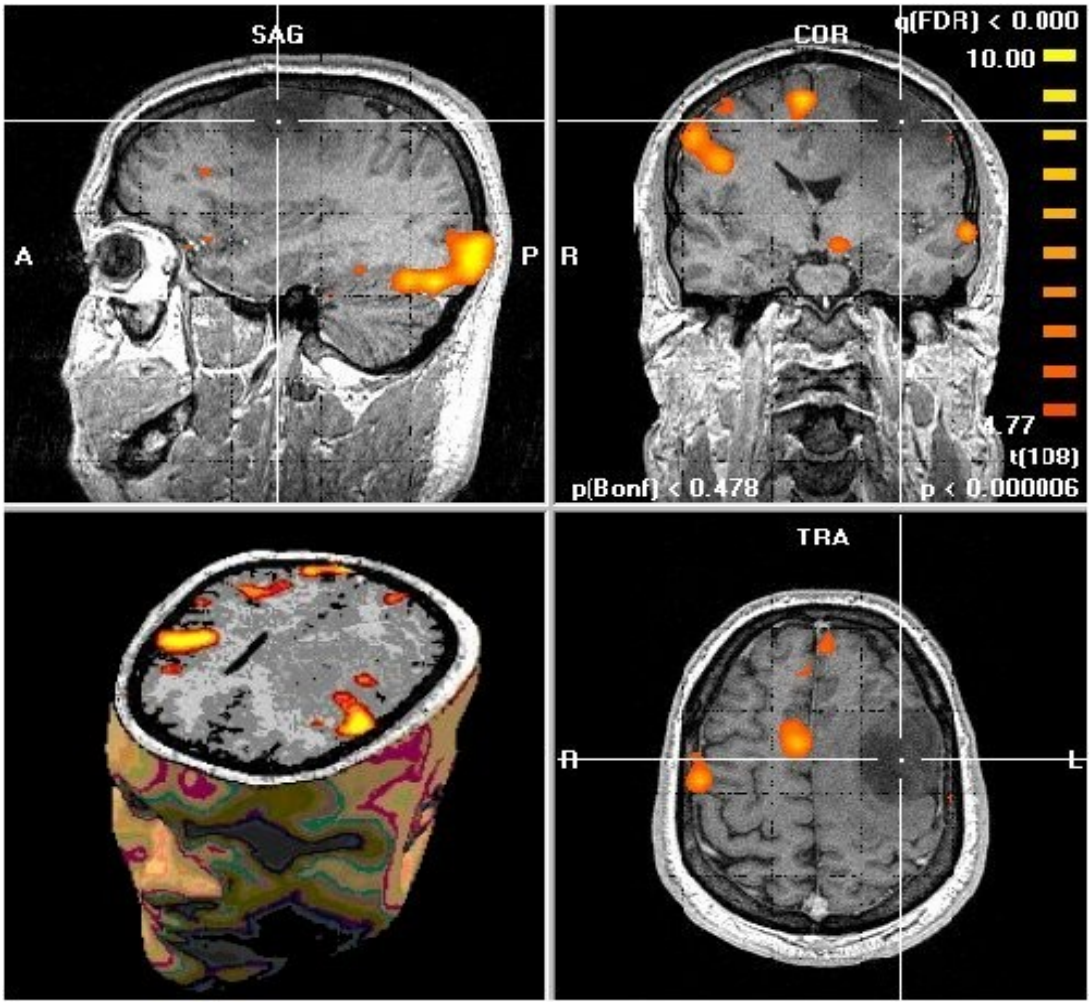

Figura 2: Ejemplo de actividad calculada.

El problema inverso es complejo, ya que a partir del campo medido en un relativamente reducido número de sensores se ha de intentar deducir la actividad completa que ha sucedido en el interior de la cabeza.

Ante la imposibilidad de calcular la solución real, se decidió utilizar un modelo simplificado del cerebro, asignando puntos en el interior del volumen que constituyen los puntos objeto de estudio (fuentes) y suponiendo que la actividad en el resto de puntos del volumen es nula.

Este problema requiere de una matriz característica (matriz *leadfield*) que representa la geometría del mallado de los sensores y la distribución de las fuentes en el volumen.

La citada matriz se calcula con la resolución del problema directo, para el que existen distintos métodos. Cada uno de ellos modela la conducción de campos por el cerebro de una fuente.

El método utilizado para calcular la matriz *leadfield* utilizada en este trabajo ha sido el método de Nolte.

Para dar solución a este problema se utiliza comúnmente el algoritmo *beamforming*, que a partir de las mediciones tomadas en los 102 sensores de la malla del casco obtiene una solución posible para los 2459 puntos del modelo del cerebro. La implementación recomendada es el *toolbox fieldtrip*, desarrollado en MATLAB.

Como alternativa al uso exclusivo del mencionado algoritmo, *beamforming*, se proponen dos posibles cursos de acción: Uno de ellos supone partir de los resultados obtenidos del algoritmo citado anteriormente con el objeto de mejorarlos minimizando su error mediante el uso de un algoritmo de búsqueda aleatoria. El otro supone el uso exclusivo del algoritmo de búsqueda aleatoria, en este caso la herramienta elegido a tal efecto es el algoritmo de minimización por búsqueda aleatoria Solis-Wets.

4. Algoritmo Solis-Wets

El algoritmo Solis-Wets es un algoritmo de minimización por búsqueda aleatoria desarrollado por Francisco J. Solis y Roger J-B. Wets en 1981.

Ese algoritmo se estructura en dos tramos, la evolución de la solución y la función de evaluación de la solución.

Evolución de la solución

El algoritmo Solis-Wets requiere de una solución inicial que va modificándose en mayor o menor medida durante su ejecución gracias a una variable *bias* y una desviación aleatoria que, combinadas con la solución actual, producen la nueva solución para su evaluación.

Generación de la nueva solución:

$$S' = f(S, bias, \text{desv\_aleatoria})$$

Posteriormente la variable *bias* se actualiza en función de si que nueva solución generada sea mejor (acierto) o peor (fallo) que la que se había obtenido anterioriormente.

La desviación aleatoria es más o menos significativa en función del número de aciertos o fallos consecutivos, de tal forma que existe correlación entre ambas variables: si hay muchos fallos consecutivos, se reduce la aleatoriedad, y si hay muchos aciertos consecutivos, se incrementa.

La variable que gestiona la aleatoriedad de dicha desviación se denomina rho.

La condición de parada del algoritmo está gestionada por la aleatoriedad de la desviación, si ésta es menor que un límite inferior, indica que el algoritmo ha encontrado demasiados fallos consecutivos y no es capaz de encontrar una solución mejor.

Los parámetros que recibe el algoritmo Solis-Wets son los siguientes:

- Función de generación de una desviación aleatoria en función de un parámetro rho. (Las más usuales son una distribución normal centrada en 0 y con σ = rho o una distribución uniforme en el rango [-rho,rho])
- Función de evaluación de una solución, particular para cada problema.
- Solución inicial. (x)
- Rho inicial.
- Rho mínimo que tiene que alcanzar para que se pare el algoritmo. (rho_lower_bound)
- Fallos consecutivos antes de modificar rho. (max_fail)
- Constante por la que es multiplica rho cuando se producen los fallos consecutivos. (ct_factor)
- Aciertos consecutivos antes de modificar rho. (max_succ)
- Constante por la que es multiplica rho cuando se producen los aciertos consecutivos. (ex_factor)

Pseudocódigo del algoritmo Solis-Wets:

```
best = INFTY
bias = 0
n_succ = 0
n_fail = 0

while (rho > rho_lower_bound)
  random_desv = generate(rho)
  curr = f(x + random_desv + bias)
  if (curr < best)
     bias = 0.6 * bias + 0.4 * random_desv
     x += dx + bias
     n_succ++
     n_fail=0
  else
     curr = f(x - random_desv - bias)
     if (curr < best)
        bias = 0.6 * bias - 0.4 * random_desv
        x -= dx + bias
        n_succ++
        n_fail=0
     else
        bias = bias/2
        n_fail++
        n_succ=0
     endif
  endif

  if (n_succ >= max_succ)
     n_succ=0
     rho = ex_factor *  rho
  endif
  if (n_fail >= max_fail)
     n_fail=0
     rho = ct_factor *  rho
  endif
endwhile
```

Código 1: Pseudocódigo del algoritmo Solis-Wets.

Evaluación de la solución

La evaluación de la solución es característica de cada problema. Los únicos requisitos necesarios para dicha función son, por un lado, que devuelva un número natural positivo o 0 y por otro lado, que si una solución es mejor que otra, la función de evaluación debe devolver un valor menor para la mejor solución, siendo 0 si la solución es exacta.

No hay inconveniente en que distintas soluciones tengan la misma evaluación, siempre y cuando no se pueda determinar cuál de ellas es mejor.

Puede darse el caso de que haya varias soluciones exactas, y al evaluar cualquiera de ellas, el resultado ha de ser 0.

S es mejor que S'

$$eval(S) < eval(S')$$
$$eval(S) \geq 0$$
$$eval(S') > 0$$

S es una solución exacta

$$eval(S) = 0$$

Para el problema del cálculo de fuentes, la función de evaluación de la solución consiste en una multiplicación de la matriz característica, matriz *leadfield,* que incluye la geometría del casco y la distribución de los puntos de la solución, por la mencionada solución.

Dicha multiplicación proyecta la solución sobre los sensores del casco, obteniendo las lecturas teóricas que se deberían haber medido en él.

Como resultado de la función, se devolverá la suma de las diferencias entre las lecturas reales y las teóricas al cuadrado.

$$Teórica = MatrizLeadfield \cdot Solución$$
$$Diferencia = Real - Teórica$$
$$eval(Solución) = \sum Diferencia[i]^2$$

5. Desarrollo de la herramienta

Esta sección expone las decisiones de diseño tomadas durante la creación de la aplicación.

Se ha tomado como referencia el código de la biblioteca GAEDA implementado por Daniel Molina Cabrera en C++, disponible bajo la licencia *GNU General Public License*, pero generando un programa autónomo para su evaluación.

Junto con la sección del código relacionada con la búsqueda de fuentes se han proporcionado datos que incluyen las lecturas del casco, la matriz característica, matriz *leadfield*, y la solución obtenida aplicando el algoritmo *beamforming* a estos datos.

Se ha decidido mantener la implementación observada en la biblioteca GAEDA sobre la que se explica en el pseudocódigo.

La versión del algoritmo Solis-Wets de la biblioteca GAEDA modifica la condición de parada por lo que ésta se ha implementado de forma que la herramienta continúa ejecutando hasta que:

- El error de la evaluación de la solución es menor que un límite.
- Se hayan generado una cantidad determinada de soluciones.
- Sin condición de parada.

La condición de parada es gestionada con los parámetros de llamada al programa.

Los parámetros con los que se ha implementado el algoritmo Solis-Wets coinciden también con los utilizados en la biblioteca GAEDA, estos son:

- Función de generación de una desviación aleatoria en función de un parámetro rho: Distribución normal centrada en 0 y con σ = rho.
- Solución inicial: Por defecto una solución aleatoria uniforme en el rango [-0.01,0.01].
- Rho inicial: 1.2 (Valor utilizado en la publicación del algoritmo Solis-Wets).
- Fallos consecutivos antes de modificar rho: 3 (Valor utilizado en la publicación del algoritmo Solis-Wets).
- Constante por la que es multiplica rho cuando se producen los fallos consecutivos: 0.5 (Valor utilizado en la publicación del algoritmo Solis-Wets).
- Aciertos consecutivos antes de modificar rho: 5 (Valor utilizado en la publicación del algoritmo Solis-Wets).
- Constante por la que es multiplica rho cuando se producen los aciertos consecutivos: 2 (Valor utilizado en la publicación del algoritmo Solis-Wets).

Parámetros del ejecutable final:
[-i <int>][-e <float>][-m <int>][-d <int>][-r <str>][-t <str>][-s <str>][-f <float>][-j <str>]

- -i: (Condición de parada): Número máximo de iteraciones del algoritmo, cada iteración del algoritmo incluye la generación de 10000 soluciones por defecto
- -e: (Condición de parada): Error que tiene que alcanzar la solución para que el programa termine.
- -m: Numero de soluciones que debe generar por iteración (10000 por defecto).
- -d: Semilla aleatoria inicial del algoritmo Solis-Wets (123456789 por defecto).
- -r: Nombre de los ficheros para continuar una ejecución terminada.
- -t: Nombre del fichero que contiene la matriz *leadfield.*
- -s: Nombre del fichero que contiene las lecturas de los sensores.
- -f: Inicializar la solución inicial a este valor (Por defecto, la inicializa con una solución aleatoria uniforme en el rango [-0.01,0-01])
- -j: Nombre del fichero que contiene la solución inicial. (Por defecto, genera una solución inicial aleatoria uniforme en el rango [-0.01,0-01])

De no estipularse condiciones de parada, la herramienta ejecutará constantemente.

Cuando el programa termine o sea terminado por una señal externa, guardará la mejor solución y las variables del algoritmo en ficheros terminados en .bias, .solution y .delta para su recuperación posterior.

Principales dificultades encontradas durante el desarrollo:

- Generación de números aleatorios independientes en varios hilos
- Pérdida de resolución en la variable que gestiona la aleatoriedad

5.1 Generación de números aleatorios independientes en varios hilos

Para mantener una correcta aleatoriedad, es imprescindible que los números pseudoaleatorios utilizados sean independientes. Esta funcionalidad revierte en cierta dificultad en la ejecución de código paralelo ya que en el caso de que varios hilos accedan a la misma variable semilla pueden darse dos situaciones:

- El acceso está protegido, lo que genera un cuello de botella.
- El acceso no está protegido, con lo que genera condiciones de carrera.

Ambas situaciones se muestran en la figura 3.

<table>
<tr><td colspan="2">Acceso protegido</td><td colspan="2">Acceso desprotegido</td></tr>
<tr><td colspan="2">Semilla = 42</td><td colspan="2">Semilla = 42</td></tr>
<tr><td>Hilo 0:

a =rand_r(&Semilla)
//Semilla = 37
...</td><td>Hilo 1:

NOP
//Semilla = 37
a =rand_r(&Semilla)
//Semilla = 232
...</td><td>Hilo 0:

a =rand_r(&Semilla)
//Semilla = 37</td><td>Hilo 1

a =rand_r(&Semilla)
//Semilla = 37</td></tr>
<tr><td colspan="2">Cuello de botella en el acceso a la variable Semilla</td><td colspan="2">El valor de a en ambos hilos es el mismo, cuando deberían trabajar con números aleatorios distintos.</td></tr>
</table>

Figura 3: Inconvenientes acceso a semilla aleatoria en ejecución paralela.

Para solucionar esta dificultad se ha decidido usar la combinación entre la función *rand_r*, que genera números pseudoaleatorios en función de una variable semilla, cuya dirección de memoria requiere como parámetro, y la creación de distintas semillas para los distintos hilos, de forma que el hilo maestro utiliza la semilla global y el resto de hilos, cuando se crean, utilizan como semilla inicial una propia creada a partir de la suma del índice único del hilo a la semilla maestra.

El código utilizado se muestra a continuación (Código 2).

```
#if defined(_OPENMP)
int my_id = omp_get_thread_num();
unsigned int otherSeed;
if (my_id){
      otherSeed = (*seed) + my_id;
      seed = &otherSeed;
}
#endif
```

Código 2: Solución semilla aleatoria en ejecución paralela.

5.2 Pérdida de resolución en la variable que gestiona la aleatoriedad

La variable que gestiona la aleatoriedad de la desviación de las soluciones generadas, rho, se multiplica por determinadas constantes según la secuencia de aciertos o fallos consecutivos.

Una de las constantes es menor que 1, por lo que puede darse el caso en que la precisión de dicha variable sea insuficiente. En este momento rho = 0 y permanecerá en este estado durante el resto de la ejecución.

Para solucionar este problema se han acotado los límites superior e inferior a dicha variable. En este rango la variable rho es significativa.

Los límites se calculan durante la inicialización de la clase que implementa el algoritmo Solis-Wets y se comprueban cada vez que la variable es modificada.

El supuesto reseñado en el párrafo anterior queda ilustrado en el Código 3.

```
min_rho = rho * 1E-8;
max_rho = rho * 1E+8;
```

Código 3: Límites de la variable rho.

Los factores por los que rho se multiplica han sido calculados experimentalmente.

En las gráficas mostradas a continuación muestran la comparación entre las evoluciones del error de distintas ejecuciones con la variación de los factores en las cotas consideradas.

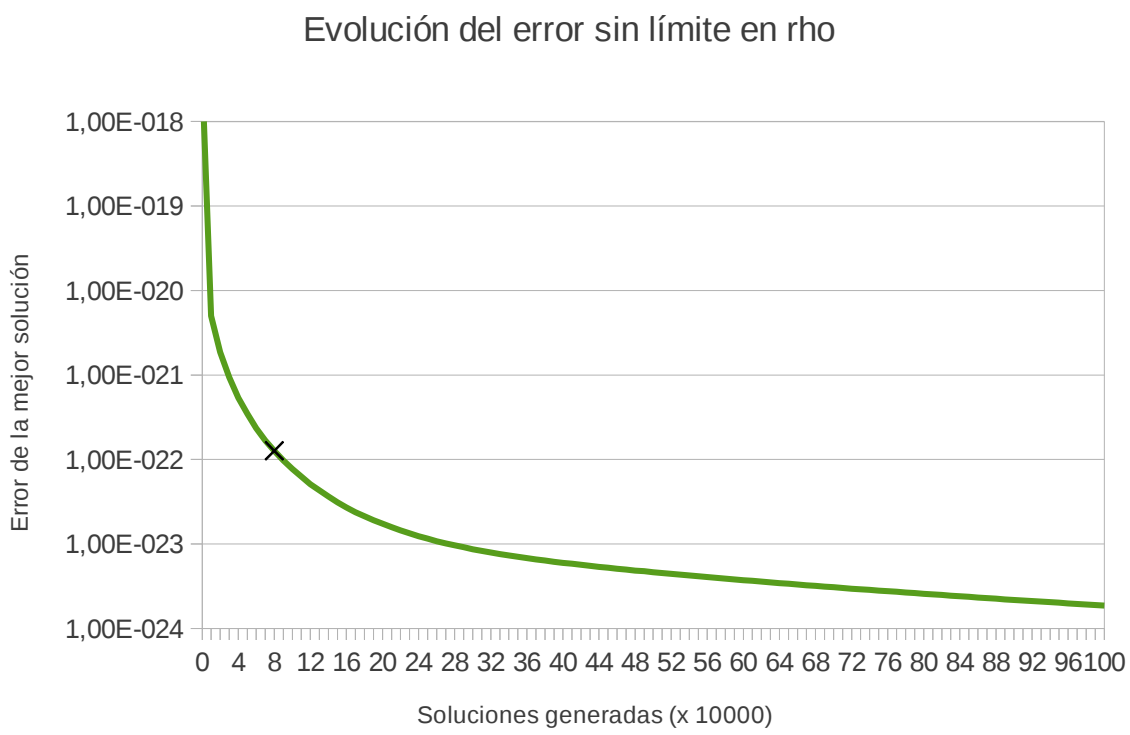

Figura 4: Evolución del error para una ejecución del programa sin límite en rho. (El eje vertical está en escala logarítmica).

En la Figura 4 puede observarse el comportamiento del error en función de la cantidad de soluciones generadas sin un límite inferior en rho. La cruz indica el punto en el que rho = 0.

Esta gráfica también muestra que el algoritmo es capaz de hacer evolucionar la solución a pesar de que rho sea 0.

Mi conjetura supone que el espacio de soluciones tiene tanta dimensiones que aún sin aleatoriedad es capaz de encontrar mejoras.

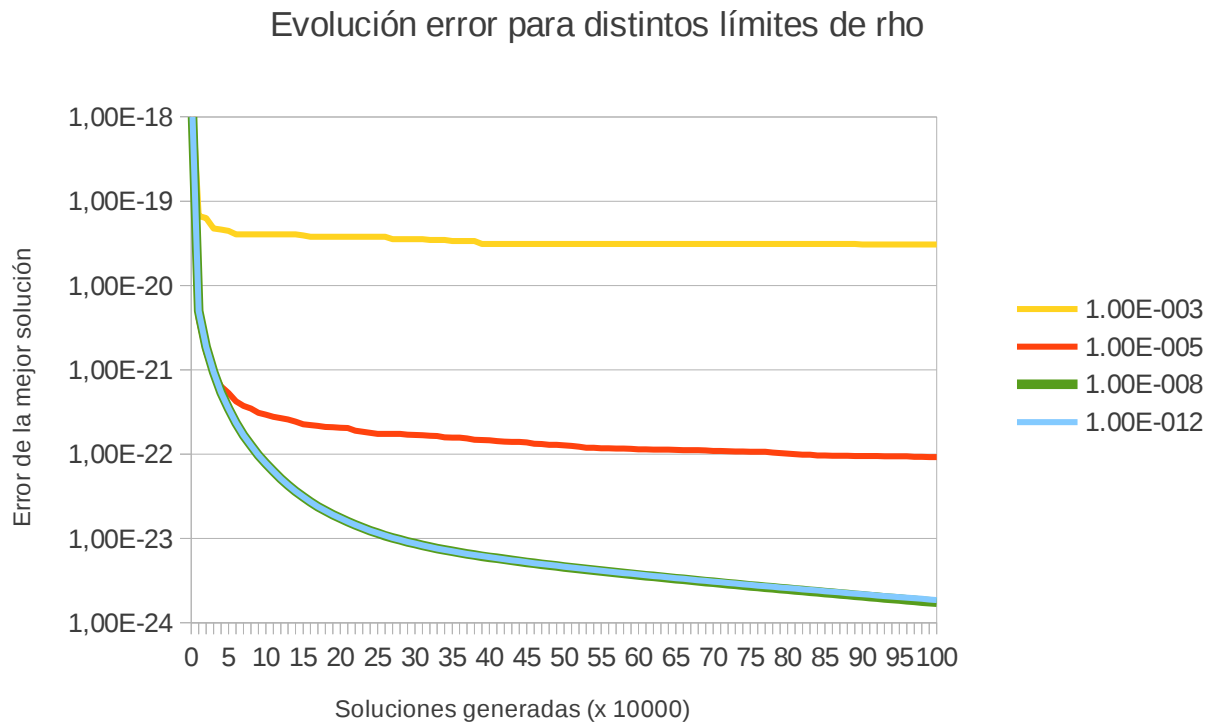

Figura 5: Evolución del error para varias ejecuciones del programa con distintos límites de rho. (La escala del eje vertical es logarítmica).

En la Figura 5 se puede apreciar cómo afecta el límite impuesto a la variable rho en el código. Puede apreciarse como en el caso de cotas 10^{-3} y 10^{3} y las cotas 10^{-5} y 10^{5} el error queda fijo en un determinado valor. Esto se debe a que con estos límites la aleatoriedad es excesiva comparada con la solución, por lo que el algoritmo Solis-Wets no consigue mejorarla, o lo hace muy lentamente.

Sin embargo, puede observarse que con las cotas 10^{8} y 10^{-8} y las cotas 10^{12} y 10^{-12} el error sigue disminuyendo durante la ejecución (Ambas gráficas se superponen). De esto se deduce que estos límites son lo suficientemente laxos como para mantener la aleatoriedad del algoritmo pero cumplen su función de evitar la pérdida de resolución.

Cabe destacar que tanto la Figura 4 como la Figura 5 se corresponden con la generación de 10^{6} soluciones y que todas ellas se diferencian únicamente en los factores, ya que la semilla aleatoria es la misma.

6. Optimización de la velocidad de generación de soluciones

Esta sección expone los distintos pasos y técnicas que se han aplicado la herramienta para acelerar la producción de soluciones.

Se han generado distintas versiones del código. La primera de ellas es muy semejante al código de GAEDA. Esta versión se ha optimizado en velocidad de generación de soluciones.

A continuación se muestran los resultados de la ejecución del programa *valgrind* con la primera versión:

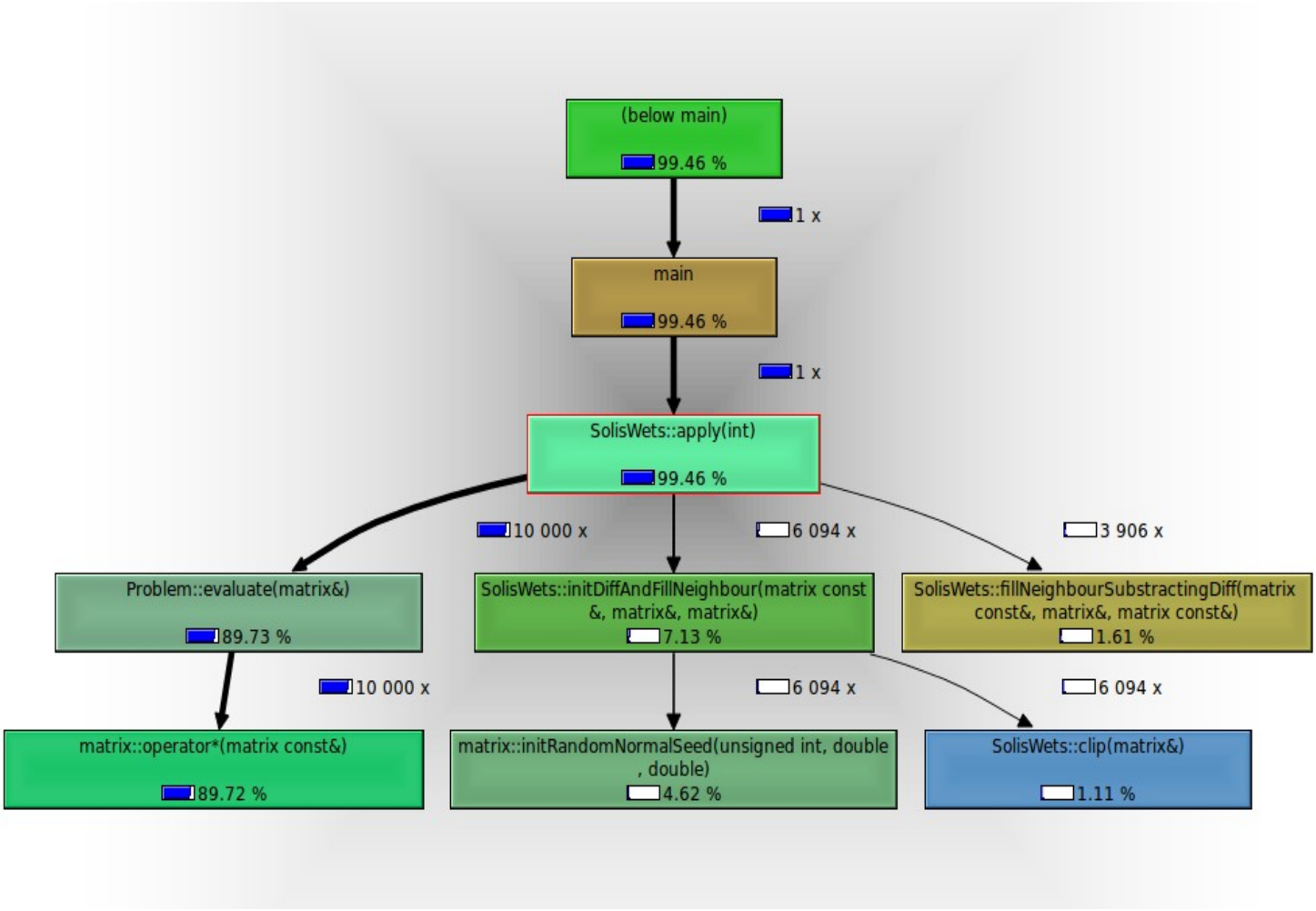

Figura 6: Resultados de la ejecución del programa valgrind con la primera versión del ejecutable.

En la figura 6 se observan los resultados de la aplicación del programa *valgrind* a la ejecución de la primera versión de la herramienta para comprobar la inversión de tiempo en las distintas funciones, habiendo generado 10^4 soluciones (*matrix*).

Como apreciarse que la evaluación de las soluciones supone un 89.73% del tiempo, en tanto que la generación de números aleatorios invierte un 4.62% del mismo. Será en ambos casos donde se concentren los esfuerzos de la optimización de la herramienta.

Las optimizaciones consideradas han sido:

- Simplificación en la generación de números aleatorios.
- Simplificación de la función de evaluación.
- Reducción de copias de las soluciones.
- Uso de distintas precisiones para los datos.
- Uso de distintas bibliotecas matemáticas y distintos compiladores.
- Paralelización sobre CPU
- Uso de la vectorización.
- Reducción de las divisiones del código.
- Paralelización sobre GPU.

6.1 Simplificación en la generación de números pseudoaleatorios

La generación de números aleatorios requeridos para el programa requieren que estos sigan una distribución normal centrada en 0 con σ variable.

Para generar dichos números aleatorios a partir de los que proporciona el sistema operativo, de distribución uniforme, se utiliza el método Box-Muller, que, a partir de dos números aleatorios uniformes genera dos números aleatorios normales:

$$Z_0 = \sqrt{(-2 \cdot \ln(U_0))} \cdot \cos(2 \cdot pi \cdot U_1)$$
$$Z_1 = \sqrt{(-2 \cdot \ln(U_0))} \cdot \sin(2 \cdot pi \cdot U_1)$$

Se observa que para el cálculo de ambos números hay operaciones que se repiten, de lo que se concluye que añadiendo dos variables intermedias puede reducirse el número de instrucciones que han de ser ejecutadas.

Supone un ahorro de la mitad de las operaciones logarítmicas, que son particularmente costosas (2.03% del tiempo total en la primera versión).

$$temp_0 = \sqrt{(-2 \cdot \ln(U_0))}$$
$$temp_1 = 2 \cdot pi \cdot U_1$$
$$Z_0 = temp_0 \cdot \cos(temp_1)$$
$$Z_1 = temp_0 \cdot \sin(temp_1)$$

El rendimiento de esta optimización se puede apreciarse en la gráfica que se muestra a continuación (Figura 7)

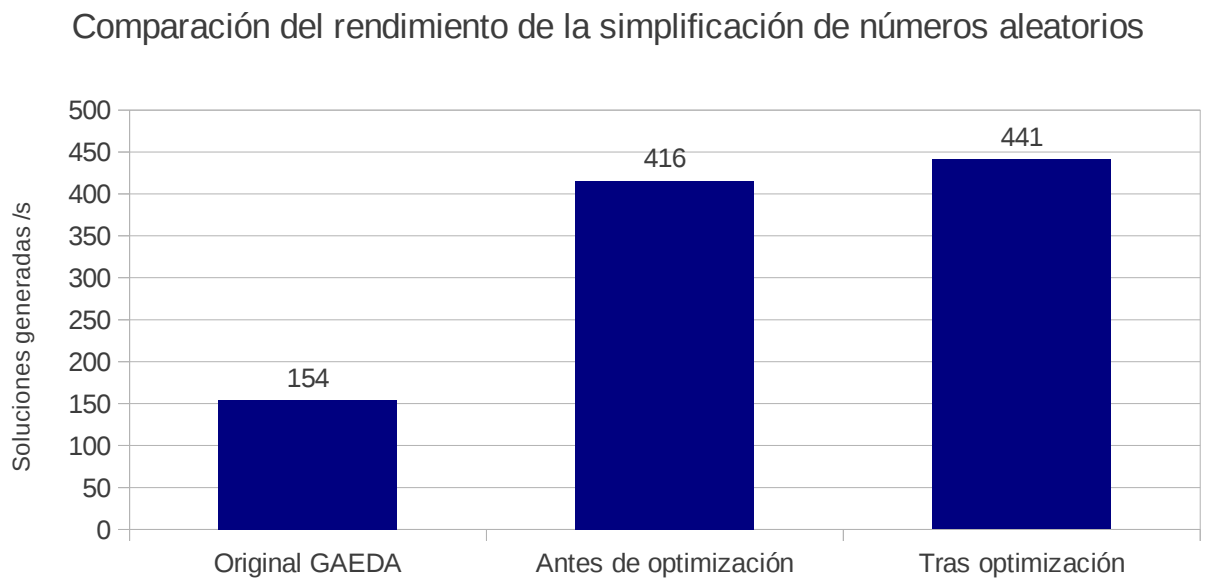

Figura 7: Comparación del rendimiento de la simplificación de la generación de números aleatorios. Prueba secuencial compilada con g++ v4.4 incluyendo -O3 y realizada en la máquina espino, equipada con 2 Intel Xeon E5645. (Exceptuando los datos de GAEDA original, que han sido proporcionados).

En la imagen anterior (Figura 7) se observa un incremento notable del rendimiento de la herramienta tras la optimización aplicada, mejora que puede atribuirse a la simplificación explicada anteriormente así como la eliminaron del código de operaciones innecesarias que se realizaban con las semillas aleatorias en la aplicación previa a la optimización.

6.2 Simplificación de la función de evaluación

La función de evaluación original multiplica la matriz por la solución (vector) y almacena el resultado en un vector intermedio que contiene las lecturas teóricas del casco. Posteriormente calcula la diferencia entre dicho vector y las lecturas reales del casco y, finalmente, acumula los cuadrados del resultado devolviendo el valor acumulado.

La optimización en este caso ha consistido en la eliminación de la variable intermedia, dado que no es utilizada posteriormente, eliminando de este modo el sobrecoste que incluye la inicialización y destrucción de dicha variable intermedia y la necesidad de recorrer todos sus valores.

El objetivo principal de esta optimización es simplificar el código de cara a otras posteriores. Esta optimización no puede aplicarse en los casos que usan bibliotecas matemáticas dado que dichas bibliotecas proporcionan funciones indivisibles.

Los resultados se muestran en las figuras expuestas a continuación (Código 3 y Código 4)

$$Teórica = MatrizLeadfield \cdot Solución$$

```
Teórica = {}
For i=1,filasM
      Acumulador = 0
      For j = 1,columnasM
            Acumulador = Acumulador + M[i,j] * Solución[j]
      End
      Teórica[i] = Acumulador
End
```

$$Diferencia = Real - Teórica$$

```
Diferencia = {}
For i = 1,filasTeorica
       Diferencia[i] = Teórica[i] – Real[i]
End
```

$$eval(Solución) = \sum Diferencia[i]^2$$

```
Resultado = 0
For i=1,filasDiferencia
      Resultado = Resultado + Diferencia[i]²
End
```

Código 3: Función de evaluación original.

```
Resultado = 0
For i=1,filasM
      Acumulador = 0
      For j = 1,columnasM
            Acumulador = Acumulador + M[i,j] * Solución[j]
      End
      Diferencia = Acumulador – Real[i]
      Resultado = Resultado + Diferencia[i]²
End
```

Código 4: Función de evaluación simplificada.

La optimización consiste en la sustitución de Código 3 por Código 4.

No se aprecia una mejoría significativa del rendimiento tras su implementación.

6.3 Reducción de copias de las soluciones

Cuando una solución es mejor que otra, el algoritmo almacena la mejor y descarta la peor.

Originalmente se copian los datos de la solución mejor sobre la peor.

Tras la optimización, las solución posee un método para reinicializarse y el programa maneja dos punteros, uno apuntando a la solución mejor y otro a la solución que se está evaluando en ese momento. De esta forma descartar la solución actual y copiar la nueva se convierte en un intercambio de punteros, evitando de esta forma la necesidad de crear, copiar y destruir toda la solución.

La optimización se ofrece en las figuras que se muestran a continuación:

```
Solution candidate;
Solution solution;
...
solution = candidate;
```

Código 5: Copia de la mejor solución (el operador = de la clase Solution está sobrecargado para realizar la copia).

```
Solution* candidate_ptr;
Solution* solution_ptr;
Solution* tmp_ptr;
...
tmp_ptr = solution_ptr;
solution_ptr = candidate_ptr;
candidate_ptr = tmp_ptr;
```

Código 6: Intercambio del puntero que apunta a la mejor solución.

La optimización consiste en la sustitución de Código 5 por Código 6.

Se aprecia una ligera mejoría del rendimiento tras su implementación, como puede apreciarse en la Figura 8.

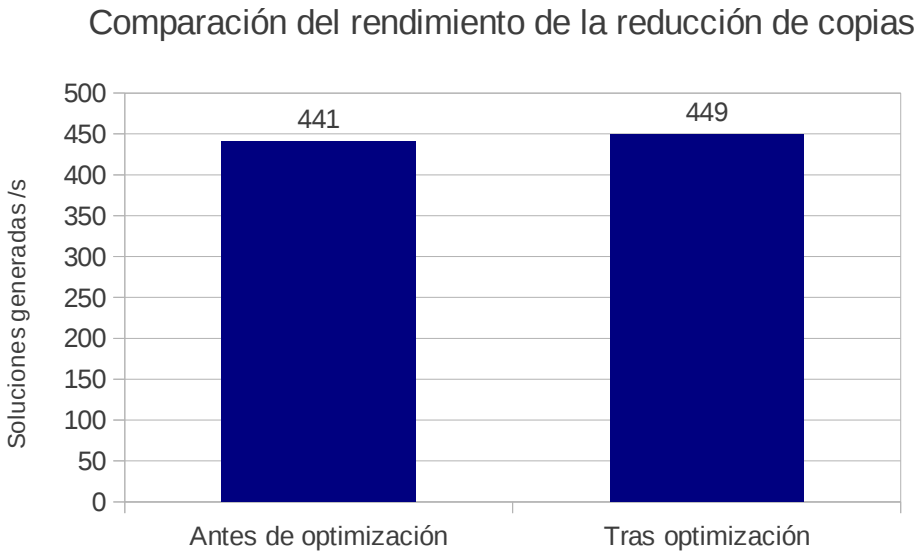

Figura 8: Comparación del rendimiento de la reducción de las copias de las soluciones. Prueba secuencial compilada con g++ v4.4 incluyendo -O3 y realizada en la máquina espino, equipada con 2 Intel Xeon E5645.

6.4 Empleo de distintas precisiones para los datos

Originalmente los datos se manejan en el formato de coma flotante en cuádruple precisión, pero las unidades funcionales de la CPU trabajan más rápido con datos en coma flotante cuando menor es su precisión, por lo que se ha comprobado si al reducir la precisión de los datos, esto afecta a los resultados obtenidos.

Los resultados obtenidos se muestran en la gráfica siguiente (Figura 9)

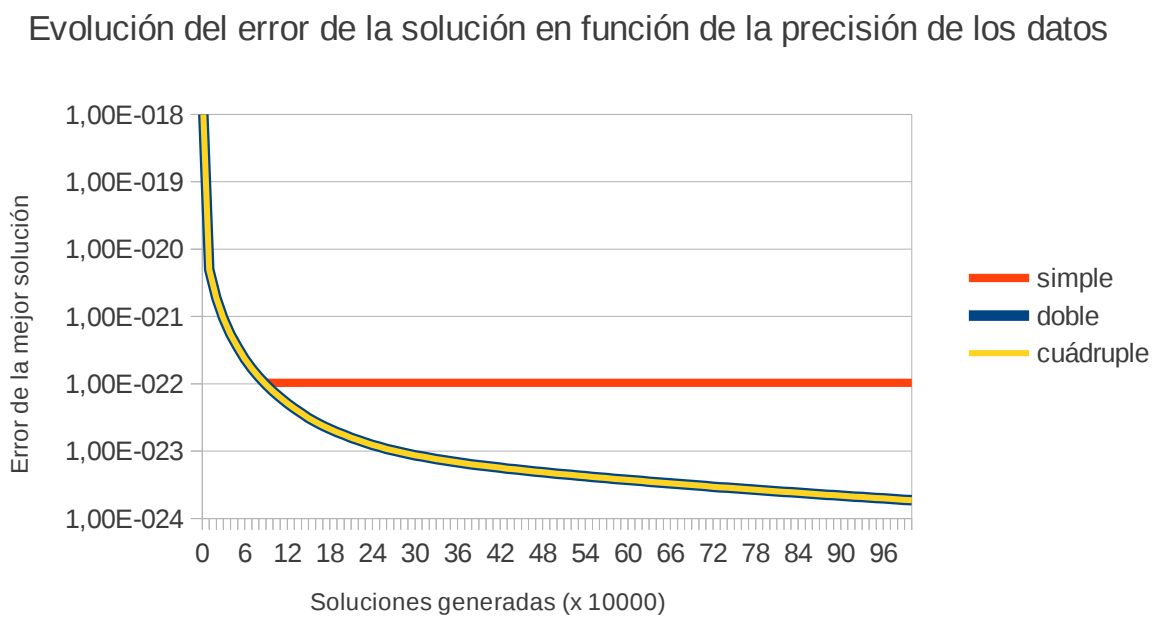

Figura 9: Evolución del error de la solución en función de la precisión de los datos durante la obtención de 10^6 soluciones. (La escala del eje vertical es logarítmica).

La Figura 9 muestra el error de la mejor solución tras haber generado una cantidad de soluciones determinada utilizando la herramienta desarrollada. Todas las series corresponden al mismo código con la misma semilla aleatoria, diferenciándose únicamente en la precisión de los datos.

Se comprueba que la simple precisión no basta para reducir el margen de error de la solución ya que éste alcanza una asíntota y no mejora.

Esto se debe a que la variable *bias* adopta valores tan reducidos que se convierten en 0, por lo que la herramienta no es capaz de hacer evolucionar la solución.

Sin embargo, no se aprecia diferencia entre los resultados entre los datos en doble y cuádruple precisión, siendo las operaciones en doble precisión considerablemente más rápidas que en cuádruple precisión, como se puede apreciar en la Figura 10.

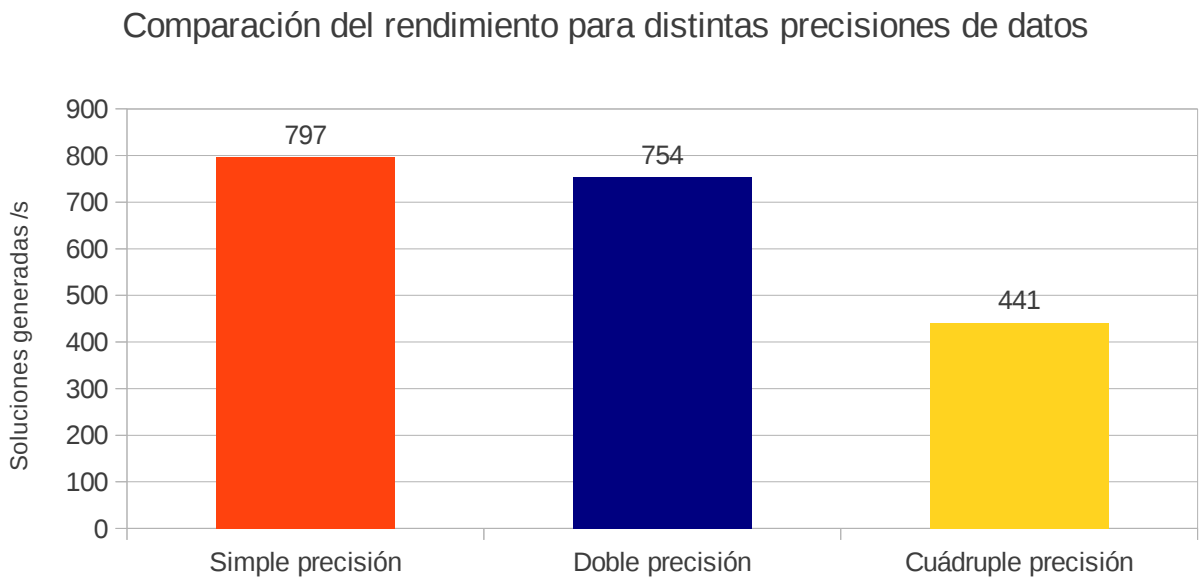

Figura 10: Comparación del rendimiento de la ejecución del programa en secuencial según la precisión de los datos. Prueba compilada con g++ v4.4 incluyendo -O3 y realizada en la máquina espino, equipada con 2 Intel Xeon E5645.

En la figura 10 se observa que las operaciones en simple precisión son las más rápidas No obstante dicha precisión no es suficiente.

Hay que destacar que la ejecución en doble precisión resulta ser un 70% más rápida que en cuádruple precisión, mientras que los resultados no reflejan diferencias.

De estos resultados se deduce que la cuádruple precisión es innecesaria, adoptando en consecuencia la doble precisión como precisión general de la herramienta.

6.5 Empleo de distintas bibliotecas matemáticas y distintos compiladores

Existen bibliotecas especializadas para cálculos matemáticos con matrices que se ajustan a este problema.

Las bibliotecas tienen distinto rendimiento, habiéndose elegido las siguientes:

- BOOST: uBLAS: Biblioteca matemática para trabajo con matrices en c++. Esta biblioteca diferencia entre vector y matriz, por lo que se han generado versiones para comprobar la diferencia de rendimiento en ambos casos. Como detalle adicional, el compilador requiere los *flags* -DNDEBUG y -DBOOST_UBLAS_NDEBUG
- Eigen: Biblioteca matemática para trabajo con matrices en c++ enfocada en el rendimiento del código. Esta biblioteca no requiere instalación ya que todo el código necesario se incluye por medio de cabeceras.
- mkl: Biblioteca matemática para c++ diseñada por Intel. Esta biblioteca incluye tanto versiones secuenciales como paralelas.

Al utilizar los *flags* de máxima optimización, distintos compiladores realizan distintas optimizaciones o las aplican mejor, por lo que se ha decidido probar con dos compiladores distintos:

- g++ versión 4.4.3
- icc versión 12.1.2

Los resultados de la comparación de las distintas bibliotecas y compiladores se muestran en las siguientes figuras (Figuras 11, 12 y 13)

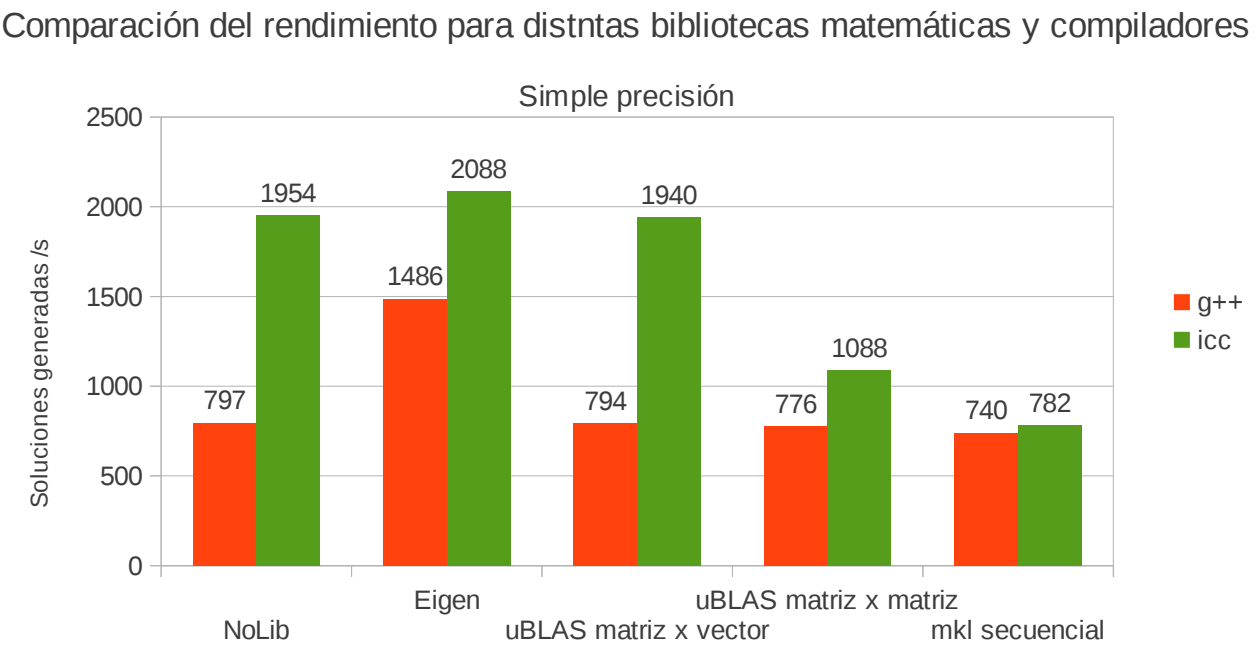

Figura 11: Comparación del rendimiento de la ejecución secuencial utilizando las distintas bibliotecas matemáticas y compiladores en simple precisión. Prueba compilada incluyendo el flag de optimización -O3 y realizada en la máquina espino, equipada con 2 Intel Xeon E5645.

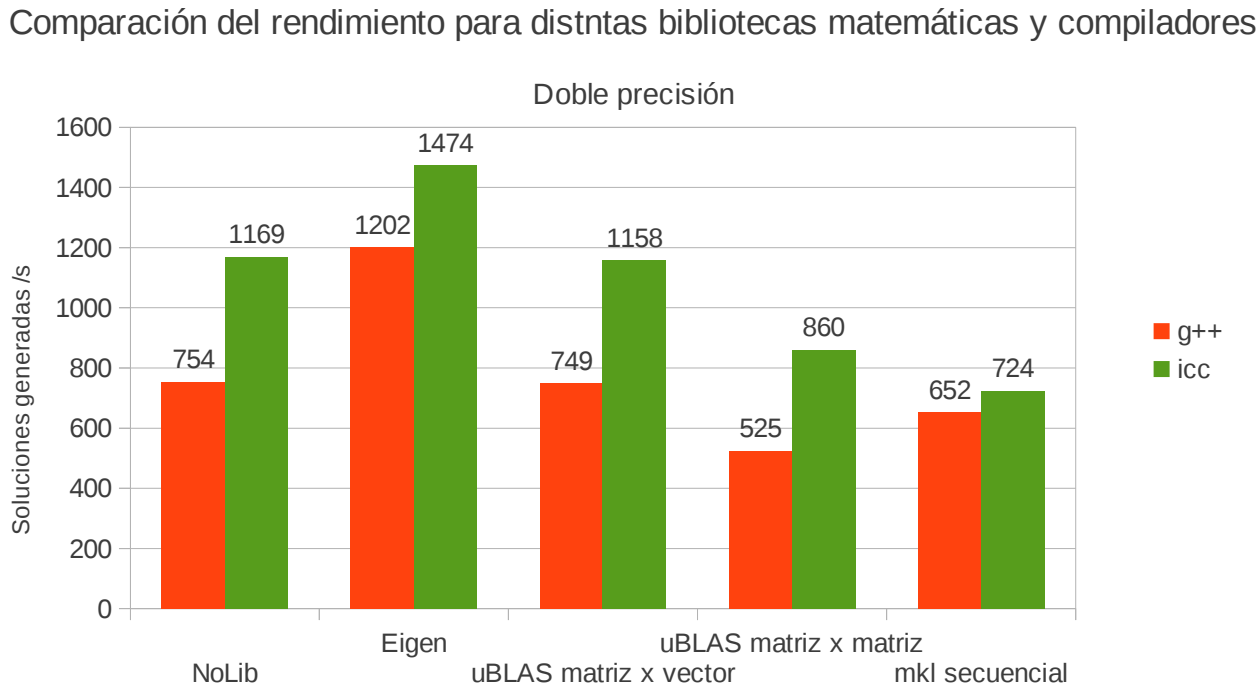

Figura 12: Comparación del rendimiento de la ejecución secuencial utilizando las distintas bibliotecas matemáticas y compiladores en doble precisión. Prueba compilada incluyendo el flag de optimización -O3 y realizada en la máquina espino, equipada con 2 Intel Xeon E5645.

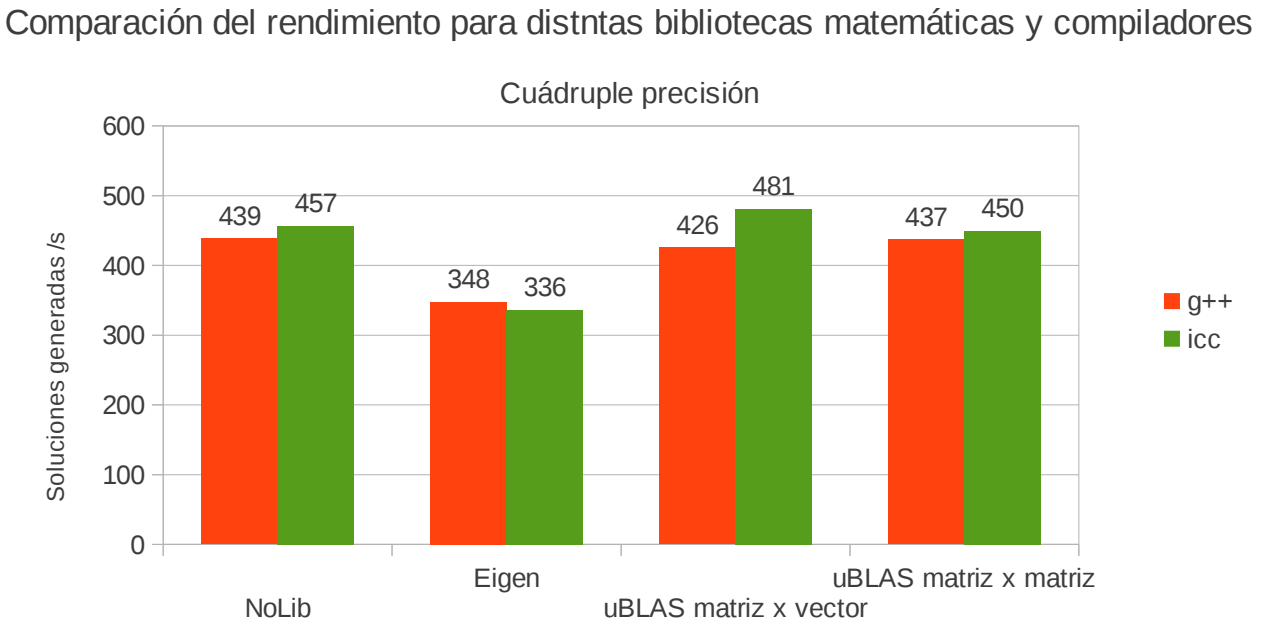

Figura 13: Comparación del rendimiento de la ejecución secuencial utilizando las distintas bibliotecas matemáticas y compiladores en cuádruple precisión. Prueba compilada incluyendo el flag de optimización -O3 y realizada en la máquina espino, equipada con 2 Intel Xeon E5645.

Como puede apreciarse en las figuras 11, 12 y 13 el rendimiento del compilador icc es superior al del compilador g++ en prácticamente todos los casos; más acusado aún en simple y doble precisión.

Asimismo puede observarse también que la biblioteca que proporciona mejor rendimiento en secuencial es la biblioteca Eigen salvo en cuádruple precisión. No obstante, para una mayor flexibilidad a la hora de modificar el código, se continuará desarrollando la versión que no utiliza ninguna biblioteca matemática (NoLib).

También se opta por descartar definitivamente las precisiones simple y cuádruple, por la insuficiente resolución de la primera, y la incapacidad de mejorar los resultados de la segunda.

Como detalle adicional, cabe destacar que la biblioteca mkl carece de instrucción para multiplicar dos matrices en cuádruple precisión.

6.6 Paralelización CPU

Con el uso de tecnología OpenMP, la paralelización de este código resulta relativamente simple debido a que sólo es necesario paralelizar las dos funciones que se exponen en la explicación de la Figura 6, que representa la generación de una desviación aleatoria y la evaluación de una solución.

Sin embargo, hay un inconveniente. La versión sin biblioteca matemática compilada con icc genera 1169 soluciones por segundo, lo que significa que invierte 855µs en generar una solución, que tiene, al menos, una evaluación, y un 61% de posibilidades de generar de desviación aleatoria.

En ese tiempo tiene que generar al menos una sección paralela y con un 61% de probabilidad, una segunda, con el sobrecoste que ello implica en una muy reducida ventana de tiempo. No se puede esperar en este caso un *speedup* similar al número de hilos, lo que hace considerar otras alternativas.

La alternativa considerada ha sido la implementación de la paralelización de un nivel superior, es decir, generar N soluciones en vez de una única y almacenar la mejor.

Según mis conjeturas, esta alternativa evolucionaría muy rápido al principio ya que se generan distintas soluciones en paralelo (el algoritmo tiene una tasa de aciertos del 37%, calculado estadísticamente), pero al final, cuando la variación aleatoria es muy pequeña, las N soluciones no serían muy diferentes, con lo que el progreso sería similar al secuencial.

Ante esta perspectiva se decidió paralelizar las funciones, asumiendo que no se logrará un *speedup* similar al número de hilos.

El resultado de lo anteriormente expuesto se muestra en la figura 14.

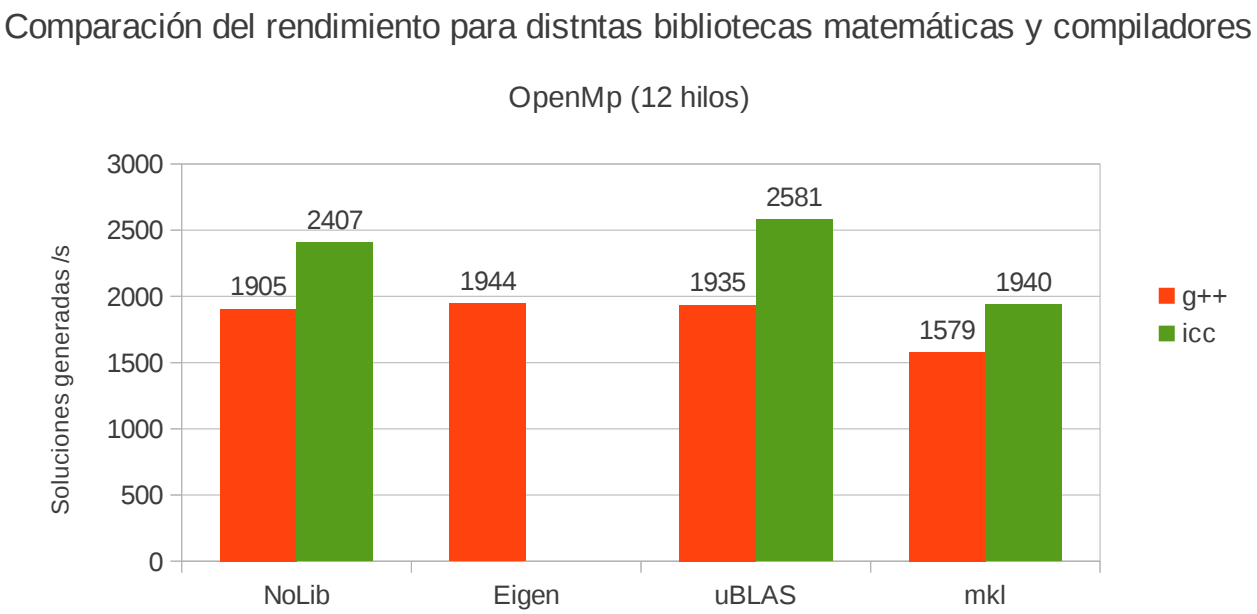

Figura 14: Comparación del rendimiento de la ejecución en paralelo (planificación OpenMP guided, solo se ha paralelizado la evaluación) utilizando las distintas bibliotecas matemáticas y compiladores en doble precisión. Prueba compilada incluyendo el flag de optimización -O3 y realizada en la máquina espino, equipada con 2 Intel Xeon E5645.

La figura 14 muestra el rendimiento similar de las distintas bibliotecas matemáticas seleccionadas salvo el de la biblioteca mkl, que es menor. El mejor de los compiladores sigue siendo mejor icc.

Existe un *bug* con la compilación de la biblioteca Eigen con OpenMP e icc v12.1.2 que ha impedido que realice la prueba.

Igualmente, la paralelización de la ejecución de la biblioteca uBLAS ha necesitado la conversión de la matriz característica en un vector de vectores fila.

Además, para realizar el producto de la matriz por la solución, cada hilo realiza una serie de productos escalares entre las filas que tenía asignadas y el vector solución. Es por eso que en la figura 14 no hay dos versiones de uBLAS.

Por último, cabe destacar que, aunque los procesadores donde se realiza la prueba permiten *HyperThreading*, se lanzan tantos hilos como procesadores físicos tiene la máquina y se configura la planificación para evitar que dos hilos sean enviados al mismo núcleo.

6.7 La vectorización y su aplicación en la herramienta

La vectorización, técnica que consiste en la combinación entre el desenrollado de un bucle y la aplicación de instrucciones SIMD (*Single Instruction Multiple Data*) por bloques a una región de memoria, permite al procesador operar con una única instrucción sobre más de un elemento al mismo tiempo, aumentando de esta forma el rendimiento general del proceso.

Existen distintos juegos de instrucciones vectoriales como pueden ser SSE, MMX o AVX.

A continuación se muestran distintas posibles configuraciones para las instrucciones vectoriales:

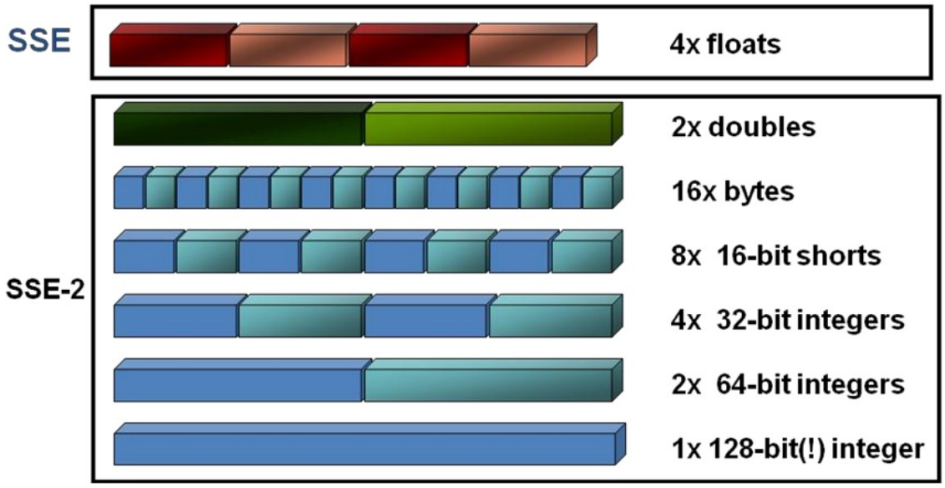

Figura 15: Una única instrucción vectorial opera sobre 128 bits. SSE2 permite distintas configuraciones de las precisiones de los datos (Imagen obtenida de la guía de vectorización de Intel).

Tras la vectorización, se incrementa el rendimiento gracias al mejor aprovechamiento de las unidades disponibles en la CPU, como muestran las figuras 16 y 17.

Figura 16: Recursos desaprovechados al operar en secuencial (Imagen obtenida de la guía de vectorización de Intel).

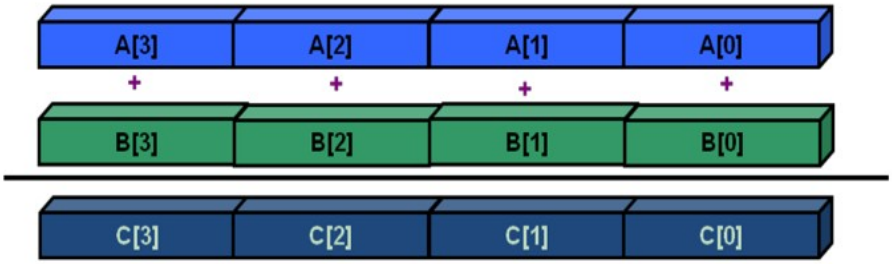

Figura 17: Recursos utilizados durante una operación vectorial (Imagen obtenida de la guía de vectorización de Intel).

El uso de estos juegos de instrucciones por parte del compilador no es trivial, ya que éste debe estar completamente seguro de que la ejecución en secuencial es equivalente a la ejecución del código vectorizado, produciendo, ante la menor duda, código secuencial.

No obstante, existen determinados *pragma* o anotaciones que se pueden incluir para indicar u obligar al compilador a que vectorice, aunque son dependientes del mismo y deben ser utilizados siempre bajo la responsabilidad del programador.

Peeling:

Para que un procesador pueda aplicar instrucciones vectoriales a unos datos, éstos deben estar alineados a memoria. Para asegurarse de que lo estén, el compilador utiliza la técnica conocida como *peeling*, consistente en procesar algunos datos del principio y del final del vector de forma secuencial de forma que el resto estén alineados a memoria.

Para evitar el sobrecoste de esta técnica, se programará de tal forma que los vectores comiencen en direcciones alineadas con instrucciones como *mm_malloc*.

Es necesario indicar al compilador que no es necesario aplicar la técnica *peeling* por medio de *flags* de compilación o anotaciones en el código.

En el código de la herramienta se utiliza una variación de esta técnica con el objeto de que, al vectorizar, dicho código no acceda a direcciones de memoria que no han sido reservadas para el proceso. Para esto debe siempre reservar espacio para el siguiente múltiplo del factor de vectorización superior o igual al tamaño necesario.

El código 7, representado a continuación, ilustra la conclusión desarrollado en le párrafo anterior.

```
 reserve(size&(FACTOR_VECTORIZE-1) ? (size&~(FACTOR_VECTORIZE-1)) +
FACTOR_VECTORIZE : size);
```

Código 7: Reserva de más espacio del necesario de forma que al vectorizar no acceda a posiciones de memoria reservadas para otras variables. Se usa la función reserve() porque la clase Solution hereda de std::vector .

Los principales problemas detectados al vectorizar el código han sido los siguientes:

- Dependencias entre los elementos del bucle.
  Si el compilador no puede asegurar la independencia de los datos, producirá código secuencial para evitar que existan condiciones de carrera.
  Se puede utilizar la palabra clave *restrict* para indicar al compilador que una variable es independiente, sin embargo, se ha decidido utilizar punteros a memoria ya que la palabra clave depende del compilador.

  Para ejemplificar este problema y su solución se muestran los códigos 8 y 9 a continuación:

```
void SolisWets::increaseBias(const Solution& diff){
   const unsigned int size = bias.size();

   for(unsigned int i=0; i< size; i++)
      bias_mem[i] = 0.6 * bias[i] + 0.4 * diff[i];
}
```

Código 8: Función de incremento de la variable bias no vectorizable.

```
void SolisWets::increaseBias(const Solution& diff){
   const unsigned int size = bias.size();

   real_t * bias_mem = &(bias[0]); //Obtener puntero a memoria del vector
   const real_t * diff_mem = &(diff[0]); // ^

   for(unsigned int i=0; i< size; i++)
      bias_mem[i] = 0.6 * bias_mem[i] + 0.4 * diff_mem[i];
}
```

Código 9: Función de incremento de la variable bias vectorizable.

- Instrucciones no vectoriales en el interior del bucle
  Si una instrucción que no posee una versión vectorial está en el interior de un bucle, el bucle no se puede vectorizar.
  Hay dos posibles soluciones:
  - La vectorización de la función no vectorial mediante el uso de anotaciones.
  - Extraer la función no vectorial fuera del bucle y ejecutarla secuencialmente:

  Para ejemplificar este problema se muestra el código 10:

```
for (int i=0; i<n_elements; i+=2){
   sigmasqrt_2logu= sigma*sqrt(-2*log((1 + rand_r(seed)/(RAND_MAX+2.0))));
   pi2v = pi2* rand_r(seed) /(RAND_MAX+1.0);

   me[i]   = (real_t)(mean + sigmasqrt_2logu * sin(pi2v));
   me[i+1] = (real_t)(mean + sigmasqrt_2logu * cos(pi2v));
}
```

Código 10: Función de generación de números aleatorios normales (método Box-Muller) no vectorizable.

El código 10 no se puede vectorizar porque la función *rand_r* no dispone de una versión vectorial.
La solución de este problema consiste en la creación de un *buffer* para ir almacenando valores de la función *rand_r* en secuencial y la vectorización del resto del bucle.
Esta solución se ilustra en el código 11:

```
for (int i=0; i<n_elements; i+=FACTOR_VECTORIZE){
    int rand_vec[FACTOR_VECTORIZE];
    for(int j=0; j<FACTOR_VECTORIZE; j++)
        rand_vec[j]=rand_r(seed);

    for(int j=0; j<FACTOR_VECTORIZE; j+=2) {
        sigmasqrt_2logu = sigma*sqrt(-2*log((1.0 + rand_vec[j])*divis));
        pi2v = rand_vec[j+1]*divis2;

        me[i+j]     = sigmasqrt_2logu * sin(pi2v);
        me[i+j+1] = sigmasqrt_2logu * cos(pi2v);
    }
}
```

Código 11: Función de generación de números aleatorios normales (Box-Muller) vectorizable.

Como puede observarse en el Código 11, ni el bucle sobre i ni el bucle que rellena el *buffer* son vectorizables, sin embargo, el tercer bucle, que es el que incluye las funciones más costosas, es vectorizable (icc es capaz de vectorizarlo, mientras que g++ v4.4 no lo es)

- Tamaño del bucle variable: Si el tamaño del bucle varía durante la ejecución del mismo, dicho bucle no se puede vectorizar.
  Basta con que el compilador no pueda asegurar que el tamaño no cambia para que no vectorice el código.
  Para ejemplificar este problema y su solución se muestran los códigos 12 y 13 a continuación:

```
for(unsigned int i=0; i< getSize(); i++)
   ...
```

Código 12: Bucle no vectorizable.

```
const unsigned int size = getSize();
for(unsigned int i=0; i< size; i++)
   ...
```

Código 13: Sustitución de Código 12 para convertirlo en vectorizable.

El código representado en Código 12 no es vectorizable porque el compilador desconoce si la función *getSize()* devuelve siempre el mismo valor, mientras que el representado en Código 13 se vectoriza sin problemas.

Durante las pruebas realizadas en el proceso de vectorización se ha comprobado que ésta es más sencilla con el compilador icc v12.1 que con g++ v4.4, siendo el algunos casos necesario modificar de gran forma el código original para que g++ lo vectorizase mientras que en el compilador de intel esta vectorización era automática, y en otros casos los intentos de vectorización han sido infructuosos con g++, como es el caso del Código 11.

También hay que destacar que el compilador icc intenta vectorizar por defecto, mientras que el g++ requiere los flags -ftree-vectorize para la vectorización general y -ffast-math si hay alguna reducción en un código vectorizado.

Dadas estas características, se considera que el compilador de intel, icc, es más adecuado que g++ para la aplicación de la vectorización.

Los resultados del la comparación del rendimiento antes y después de la vectorización se muestran en la siguiente gráfica (Figura 18)

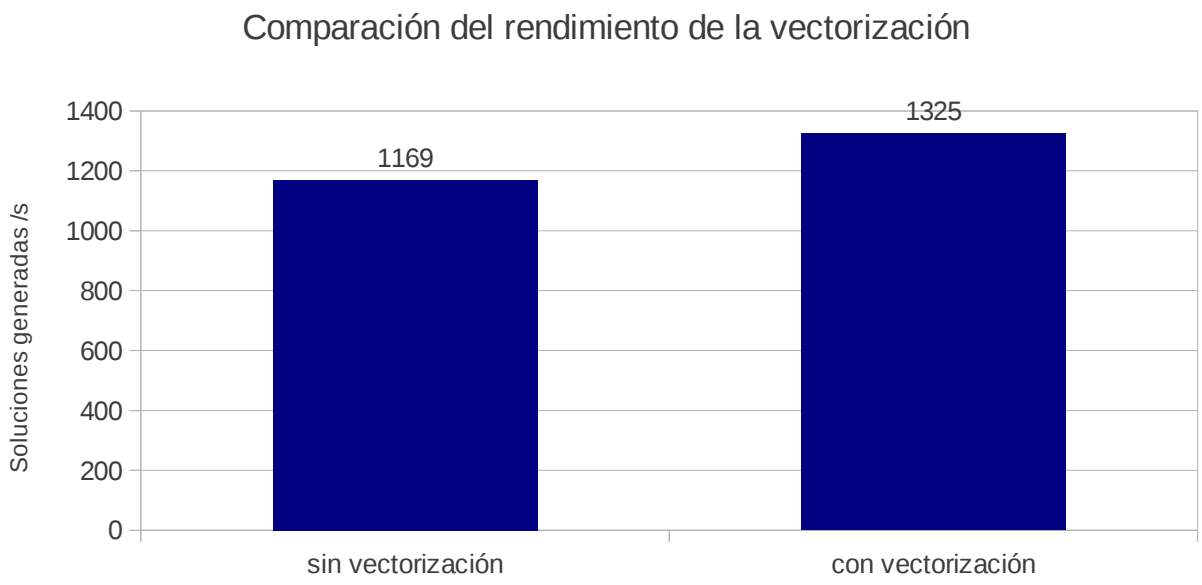

Figura 18: Rendimiento de la vectorización. Prueba secuencial compilada con icc -O3 y realizada en la máquina espino, equipada con 2 Intel Xeon E5645.

Como se aprecia en la Figura 18, el rendimiento del código vectorizado es superior. A su vez, el esfuerzo necesario para poder implementar esta optimización no es significativo, de lo que se deduce la idoneidad de la técnica.

Cabe destacar que en este trabajo la vectorización se ha aplicado a la totalidad de los bucles que trabajan sobre las soluciones.

6.8 Reducción de las divisiones en el código

Las funciones de generación de números aleatorios del sistema operativo devuelven un número entero entre 0 y la constante RAND_MAX, ambos incluidos, pero el método Box-Muller explicado anteriormente requiere $0<U_0<1,\ 0\leq U_1<1$, por lo que se ha de aplicar una conversión de los primeros en los segundos, utilizando para ello las siguientes relaciones:

$$U_0=(rand()+1)/(RANDMAX+2)$$
$$U_1=rand()/(RANDMAX+1)$$

La constante RAND_MAX no es modificada en toda la ejecución. Teniendo en cuenta, además, que un producto es más rápido que una división, se calculan dos nuevas constantes correspondientes a

$$D_0=1/(RANDMAX+2) \quad \rightarrow \quad U_0=(rand()+1)\cdot D_0$$
$$D_1=1/(RANDMAX+1) \qquad\quad U_1=rand()\cdot D_1$$

Sustituyendo una división por una multiplicación en cada generación de un nuevo número aleatorio, lo que aumenta el rendimiento de la herramienta como muestra la siguiente gráfica (Figura 19).

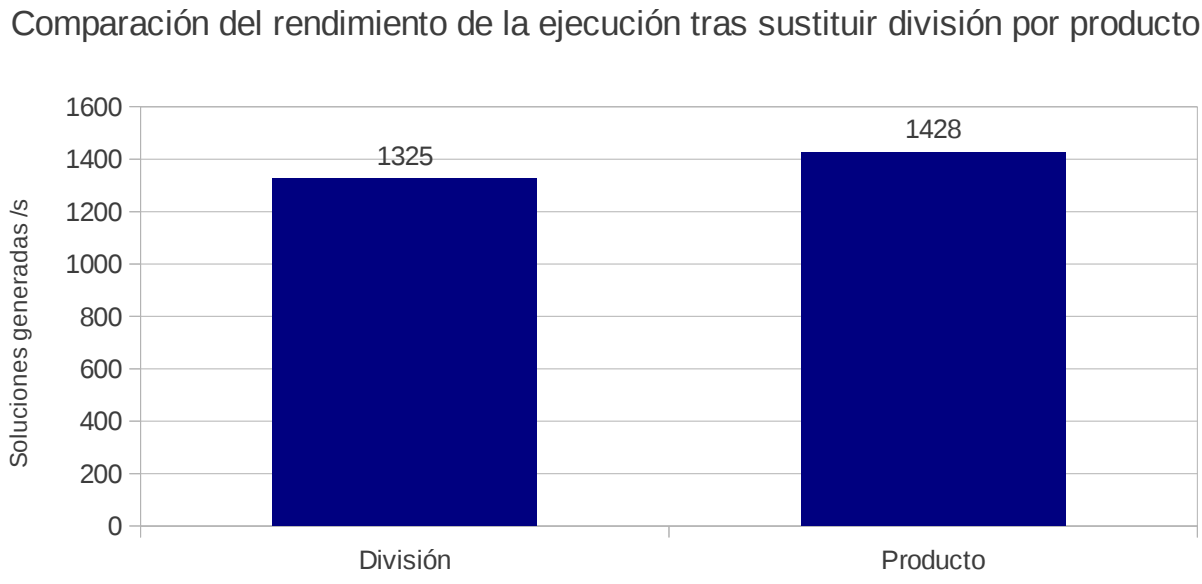

Figura 19: Rendimiento del ejecutable tras sustituir una división que se ejecutaba 7377 veces cada vez que se calcula una nueva solución por un producto. Prueba secuencial compilada con icc -O3 y realizada en la máquina espino, equipada con 2 Intel Xeon E5645.

La figura 19 muestra como un sencillo cambio que es fácil pasar por alto, como puede ser sustituir una división por un producto, puede lograr una notable mejoría en el rendimiento de la herramienta.

6.9 Resultados de la optimización

Realizadas las optimizaciones expuestas a lo largo del documento, los resultados del *valgrind* ejecutando la herramientas en secuencial quedan reflejados en la figura 20.

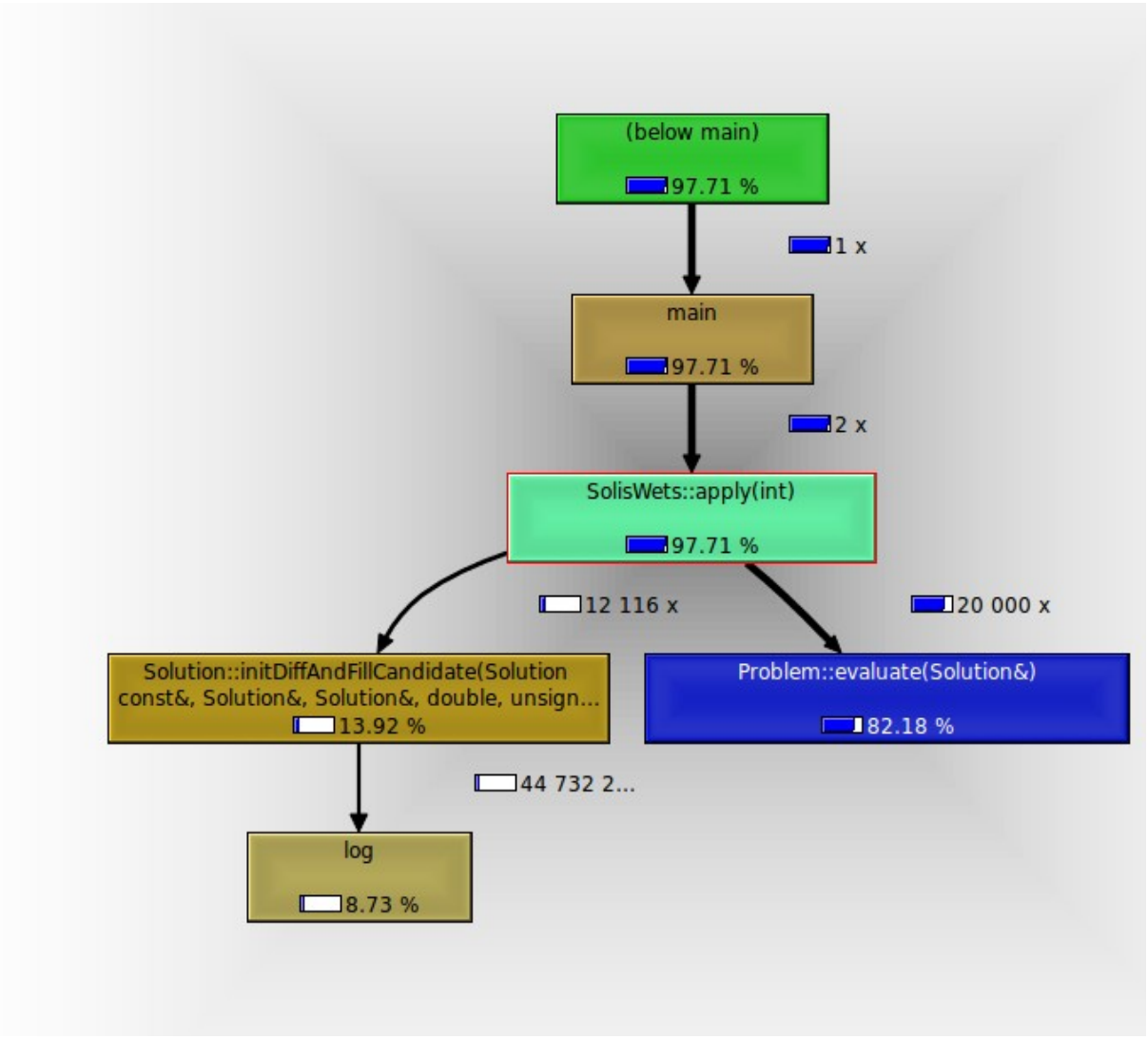

Figura 20: Resultados de la aplicación de valgrind a la ejecución de la aplicación.

En la figura 20 se aprecia la redistribución del tiempo entre las distintas funciones de la herramienta:

- El porcentaje de tiempo total empleado por la evaluación se ha reducido en un 7%.
- El porcentaje de tiempo total empleado por la inicialización de números aleatorios se ha incrementado en un 5%
- El porcentaje de tiempo total empleado por operación *log* se ha incrementado del 2% al 8.73%. El número de veces que se ejecuta esta operación se ha reducido a la mitad.

Combinando ejecución en paralelo y vectorización, obtenemos los resultados mostrados en las figuras 21 y 22:

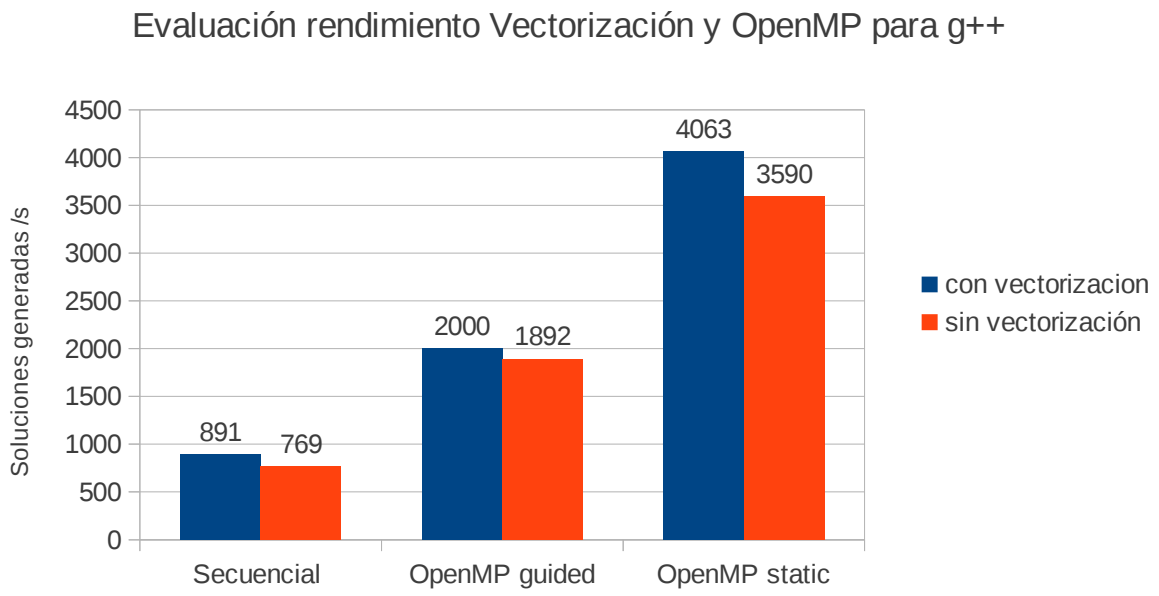

Figura 21: Rendimiento de la aplicación en secuencial y distintas planificaciones en paralelo (tanto evaluación como generación de números aleatorios) con 12 hilos. Prueba compilada con g++ incluyendo el flag de optimización -O3 y realizada en la máquina espino, equipada con 2 Intel Xeon E5645.

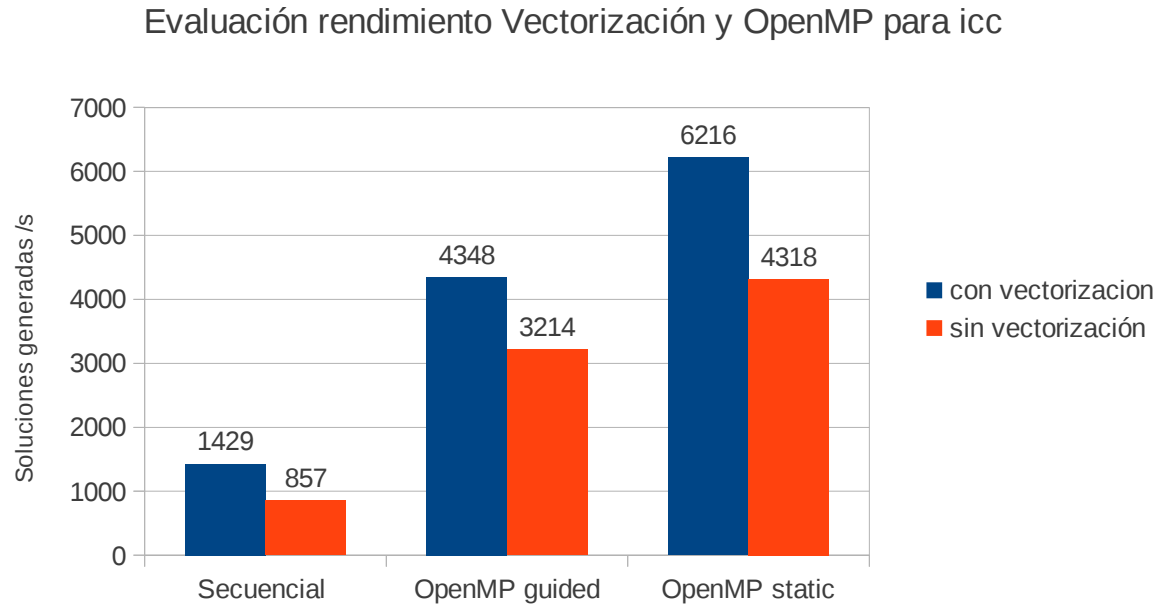

Figura 22: Rendimiento de la aplicación en secuencial y distintas planificaciones en paralelo (tanto evaluación como generación de números aleatorios) con 12 hilos. Prueba compilada con icc incluyendo el flag de optimización -O3 y realizada en la máquina espino, equipada con 2 Intel Xeon E5645.

Las figuras 21 y 22 muestran el rendimiento de la herramienta vectorizada en contraposición a la rendimiento de la herramienta sin vectorizar con g++ (Figura 21) e icc (Figura 22) para las configuraciones: secuencial, paralela con planificación *guided* y paralela con planificación *static*.

Se observa una correlación positiva entre la planificación y el rendimiento, de modo que puede interpretarse que ésta influye en gran medida en el rendimiento de la herramienta, si bien el máximo *speedup* que se alcanza es 504% a pesar de estar ejecutando 12 hilos.

El *speedup* para cada etapa de la optimización, sin incluir las bibliotecas matemáticas, se muestra en la figura 23:

Versión	Soluciones generadas/s	*Speedup* de etapa	*Speedup* Global
Inicial	416	-	-
RNG simplificado	441	1.06x	1.06x
Evitando copias de soluciones	449	1.02x	1.08x
Doble precisión	754	1.68x	1.81x
icc	1169	1.55x	2.81x
icc vectorizado	1325	1.13x	3.19x
icc vectorizado sin divisiones	1429	1.08x	3.44x
icc vectorizado sin divisiones paralelo planificación *static*	6216	4.35x	14.94x

Figura 23: Speedup logrado en cada etapa. (No incluye bibliotecas matemáticas).

En la Figura 23 se muestra la relación entre las etapas de la optimización y el *speedup* logrado en cada una de ellas.

Las mejores optimizaciones se han producido por el paso de cuádruple precisión a doble y el uso del compilador icc.

No se incluye entre las mejores optimizaciones la paralelización porque un *speedup* de 4.35x para 12 hilos, aunque esperado dadas las características del problema explicadas en la etapa de paralelización, no es suficiente.

6.10 Optimizaciones futuras

Como propuesta de continuación para optimizaciones futuras, considero interesante el estudio de la migración de toda la operación con vectores a CUDA, para trabajar con ellos sobre las GPUs.

La búsqueda de fuentes MEG tiene una característica distintiva que considero muy positiva en este aspecto: no es excesivamente complicado paralelizar masivamente el trabajo con vectores.

La circunstancia a tener en cuenta en esta propuesta de optimización futura es que el trabajo que debe realizar cada hilo es muy pequeño, por lo que es conveniente aplicar un paradigma con un gran volumen de hilos en el que el coste de su creación y destrucción sea reducido.

CUDA es la herramienta de mi elección para una aplicación futura de estas conclusiones ya que cumple las características descritas anteriormente.

7. Resultados de la ejecución de la herramienta

Esta sección incluye las soluciones obtenidas tras aplicar el programa una vez terminado a distintas soluciones de entrada.

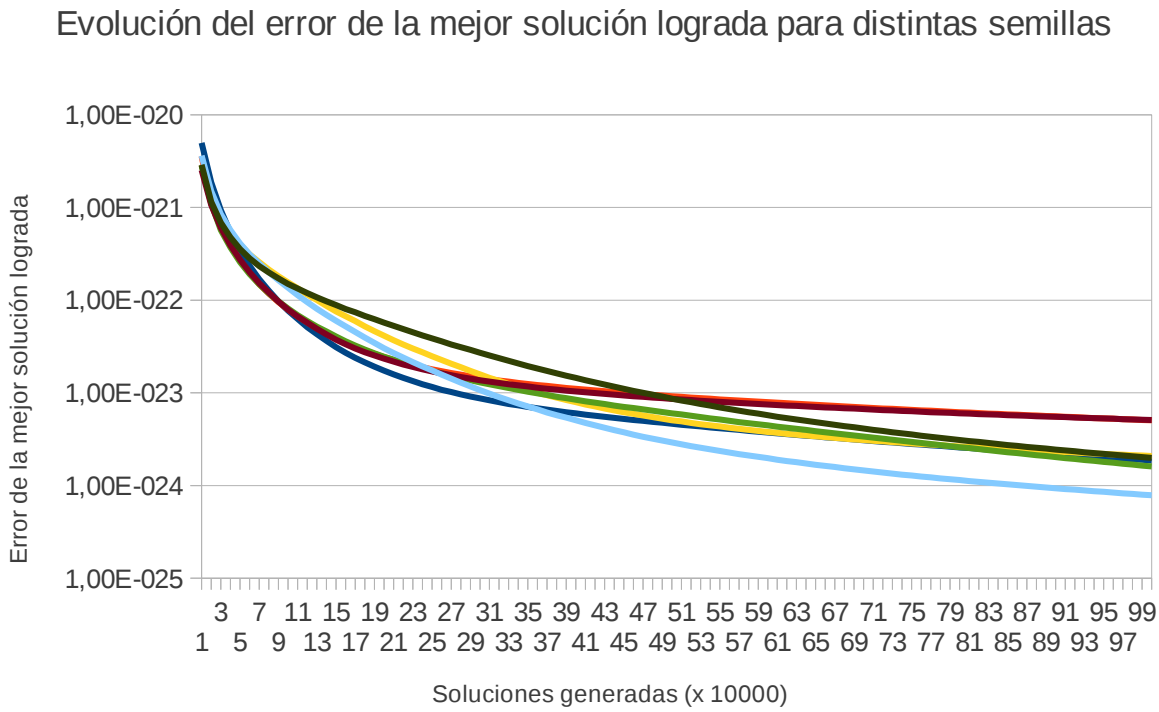

Figura 24: Evolución del error para distintas semillas aleatorias, partiendo de la misma solución inicial. (La escala del eje vertical es logarítmica).

En la Figura 24 se observa que, a pesar de tener semillas aleatorias distintas, la mayoría de las ejecuciones evolucionan de forma similar y favorable, aunque unas reducen más el error por solución generada que otras. También puede observarse que hay dos series que se evolucionan de forma ligeramente distinta.

7.1 Solución inicial aleatoria uniforme

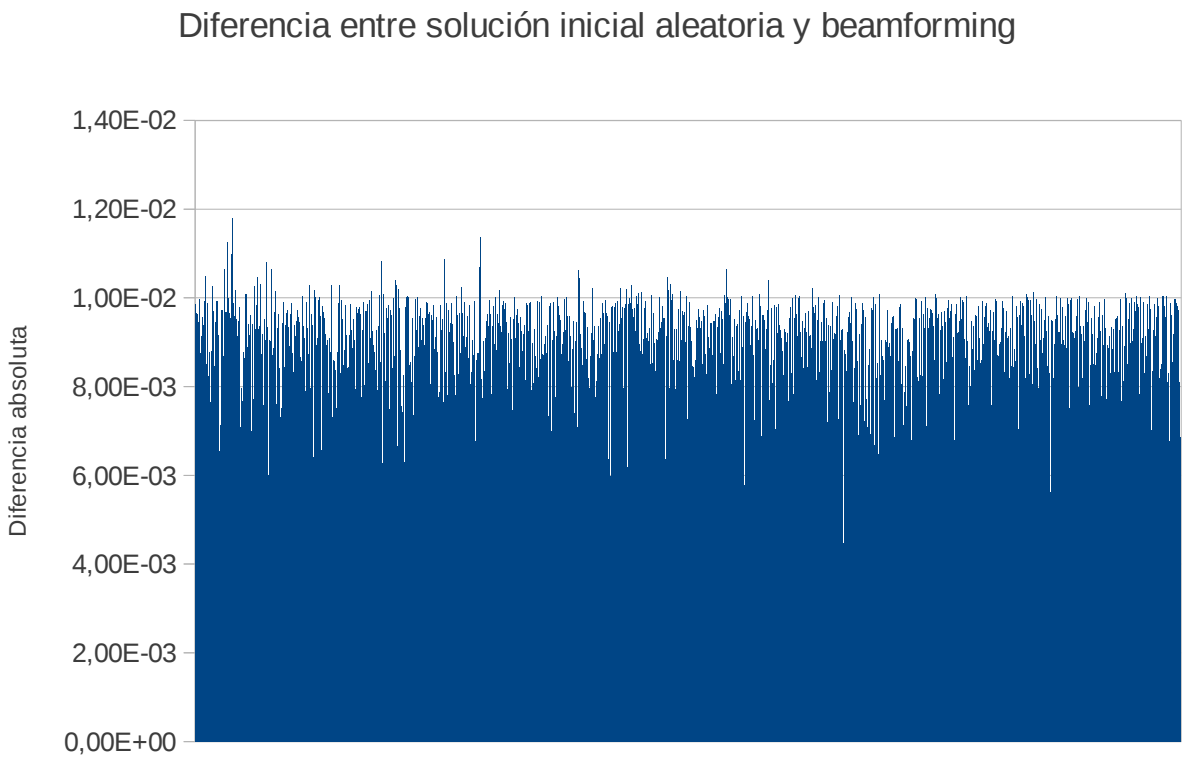

Figura 25: Diferencia absoluta entre los valores de la solución proporcionada por el algoritmo beamforming y los proporcionados por el algoritmo Solis-Wets comenzando en una solución aleatoria uniforme en el rango [-0.01,0.01] tras 10^6 soluciones generadas.

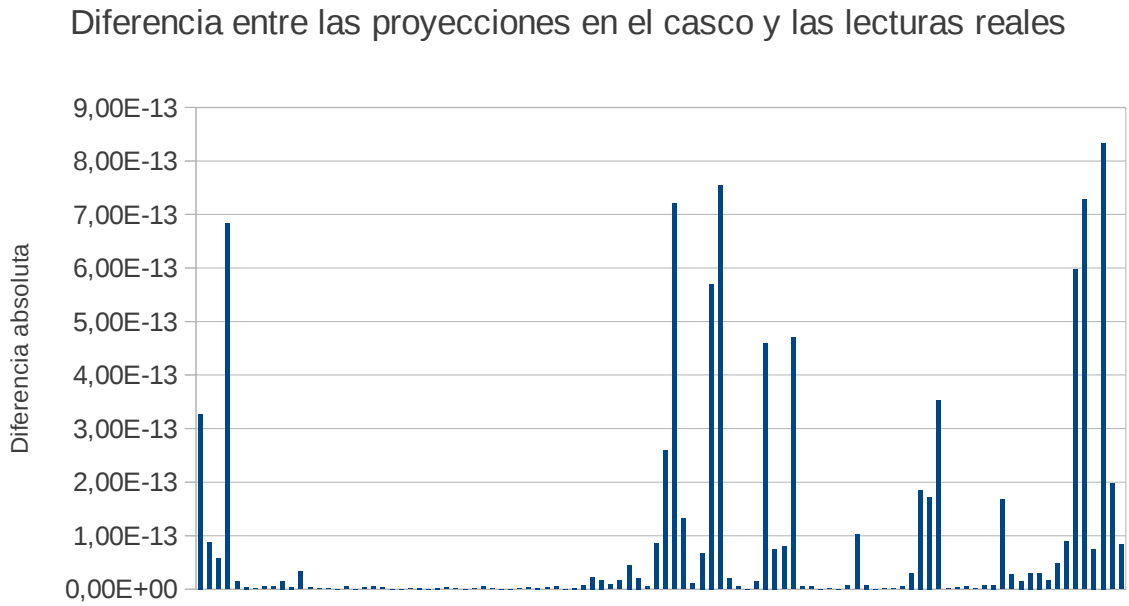

Figura 26: Diferencia absoluta entre las mediciones del casco y la proyección de la solución mediante la matriz leadfield para la solución representada en la Figura 25.

La Figura 25 muestra las diferencias en valor absoluto entre los valores de la solución proporcionada por el algoritmo *beamforming* y el algoritmo Solis-Wets habiendo empezado el último con una solución aleatoria uniforme.

La media de los valores de la figura 25 es 0.00547 y la desviación típica 0.00293.

El error de la solución proporcionada por el algoritmo Solis-Wets es $4.42*10^{-24}$

La figura 26 muestra la diferencia entre las lecturas reales del casco y las que se hubieran medido si la solución proporcionada por el algoritmo Solis-Wets fuera exacta.

La media de estos valores es $8.83*10^{-14}$, y la desviación típica, $1.89*10^{-13}$

7.2 Solución inicial proporcionada por el algoritmo *beamforming*

En esta ejecución, la solución inicial es la solución obtenida por el algoritmo *beamforming*

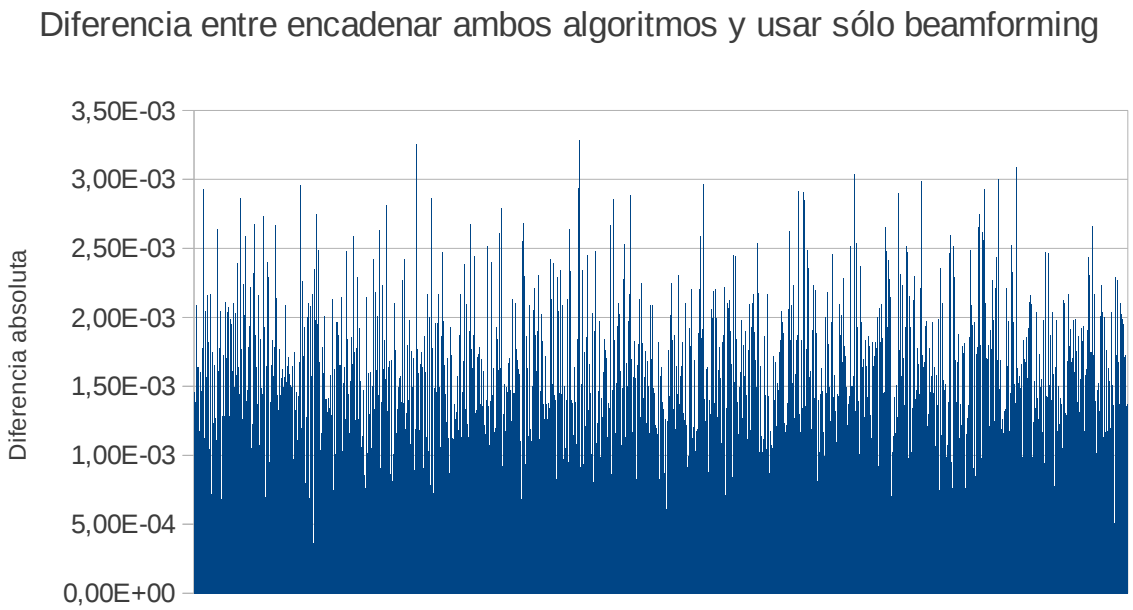

Figura 27: Diferencia absoluta entre los valores de la solución proporcionada por el algoritmo beamforming y los proporcionados por el algoritmo Solis-Wets comenzando en la solución proporcionada por el algoritmo beamforming tras 10^6 soluciones generadas.

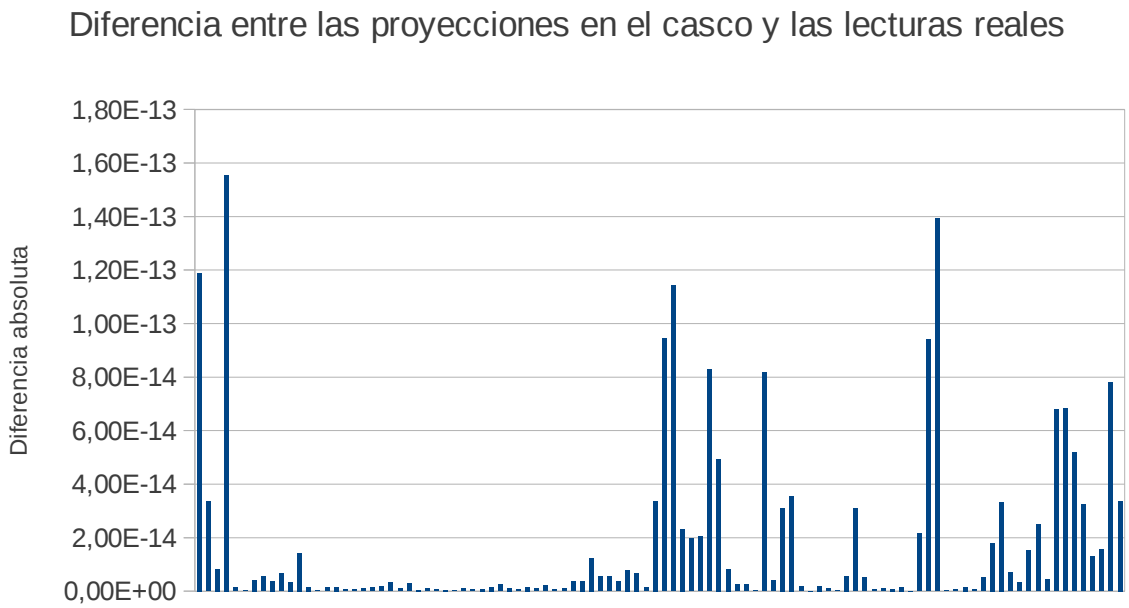

Figura 28: Diferencia absoluta entre las mediciones del casco y la proyección de la solución mediante la matriz leadfield para la solución representada en la Figura 27.

La Figura 27 muestra las diferencias en valor absoluto entre los valores de la solución proporcionada por el algoritmo *beamforming* y la solución proporciona por el aplicación secuencial de ambos algoritmos.

La media de los valores de la figura 27 es 0.000750 y la desviación típica 0.000561

El error de la solución proporcionada por el algoritmo Solis-Wets es $1.35*10^{-25}$

La figura 28 muestra la diferencia entre las lecturas reales del casco y las que se hubieran medido si la solución proporcionada por el algoritmo Solis-Wets fuera exacta.

La media de estos valores es $1.79*10^{-14}$, y la desviación típica, $3.19*10^{-14}$

7.3 Solución inicial a 0

En esta ejecución, todos los valores de la solución inicial son 0

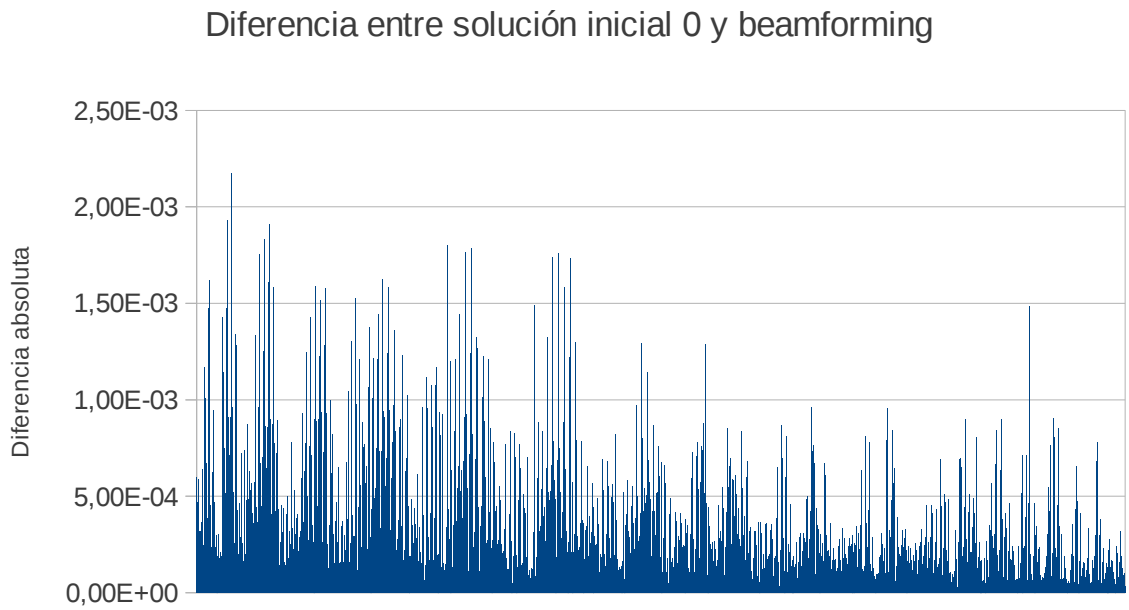

Figura 29: Diferencia absoluta entre los valores de la solución proporcionada por el algoritmo beamforming y los proporcionados por el algoritmo Solis-Wets comenzando una solución formada completamente por 0 tras 10^6 soluciones generadas.

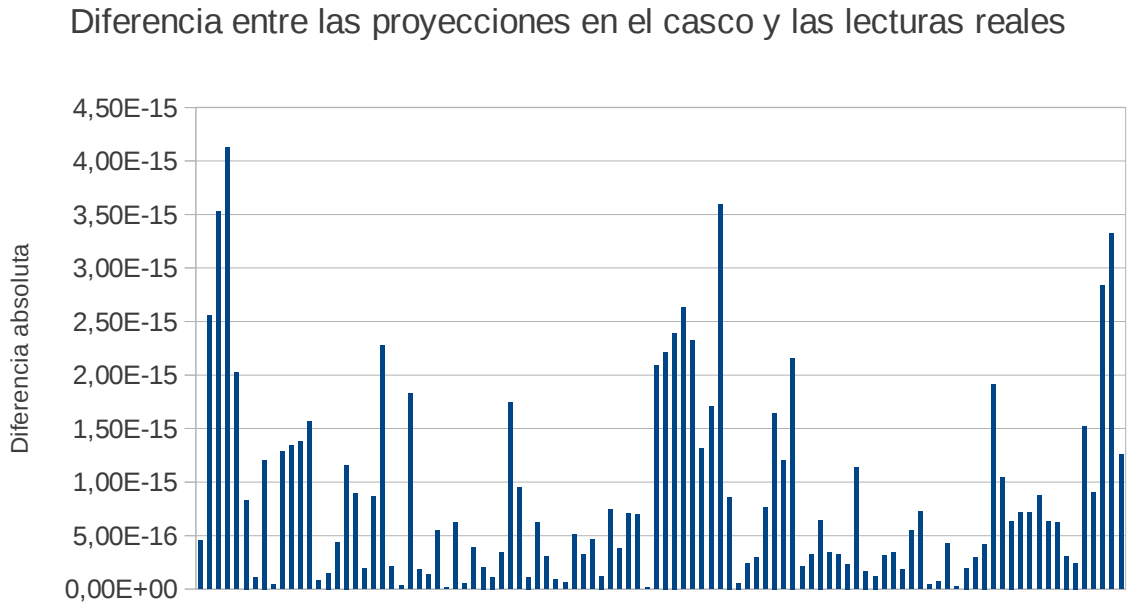

Figura 30: Diferencia absoluta entre las mediciones del casco y la proyección de la solución mediante la matriz leadfield para la solución representada en la Figura 29.

La Figura 29 muestra las diferencias en valor absoluto entre los valores de la solución proporcionada por el algoritmo *beamforming* y la proporcionada por el algoritmo Solis-Wets habiendo empleado como solución inicial solución constante 0.

La media de los valores de la figura 29 es 0.000188 y la desviación típica 0.000238

El error de la solución proporcionada por el algoritmo Solis-Wets es $1.60*10^{-28}$

La figura 30 muestra la diferencia entre las lecturas reales del casco y las que se hubieran medido si la solución proporcionada por el algoritmo Solis-Wets fuera exacta.

La media de estos valores es $8.74*10^{-16}$, y la desviación típica, $9.03*10^{-16}$

7.4 Solución inicial a $2.29*10^{-7}$ (Promedio de una solución comenzada en 0)

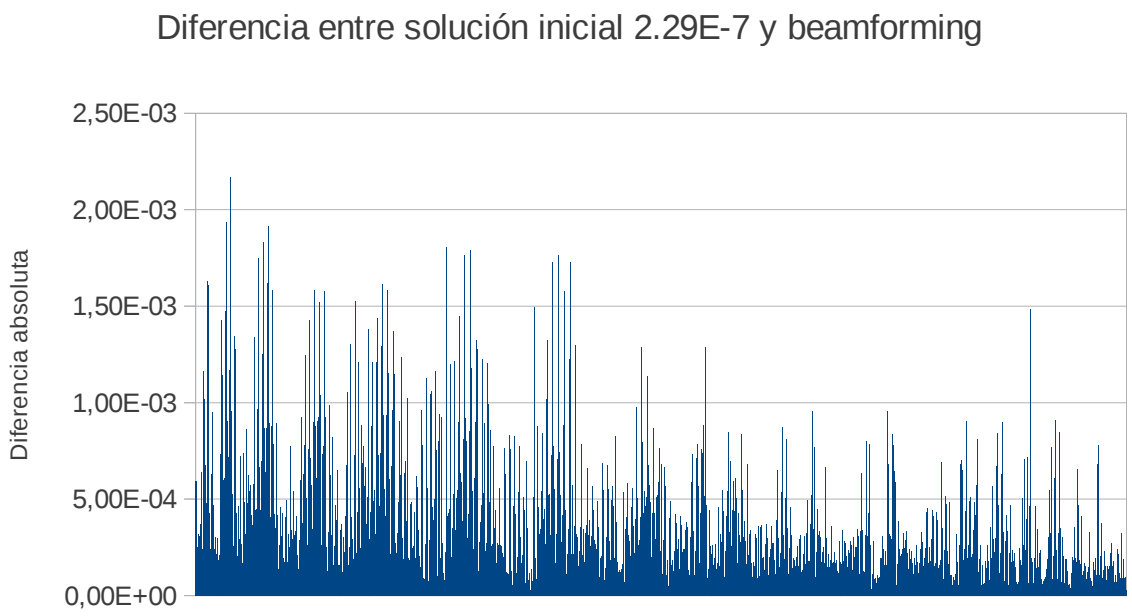

Figura 31: Diferencia absoluta entre los valores de la solución proporcionada por el algoritmo beamforming y los proporcionados por el algoritmo Solis-Wets comenzando una solución formada completamente por $2.29*10^{-7}$ tras 10^6 soluciones generadas.

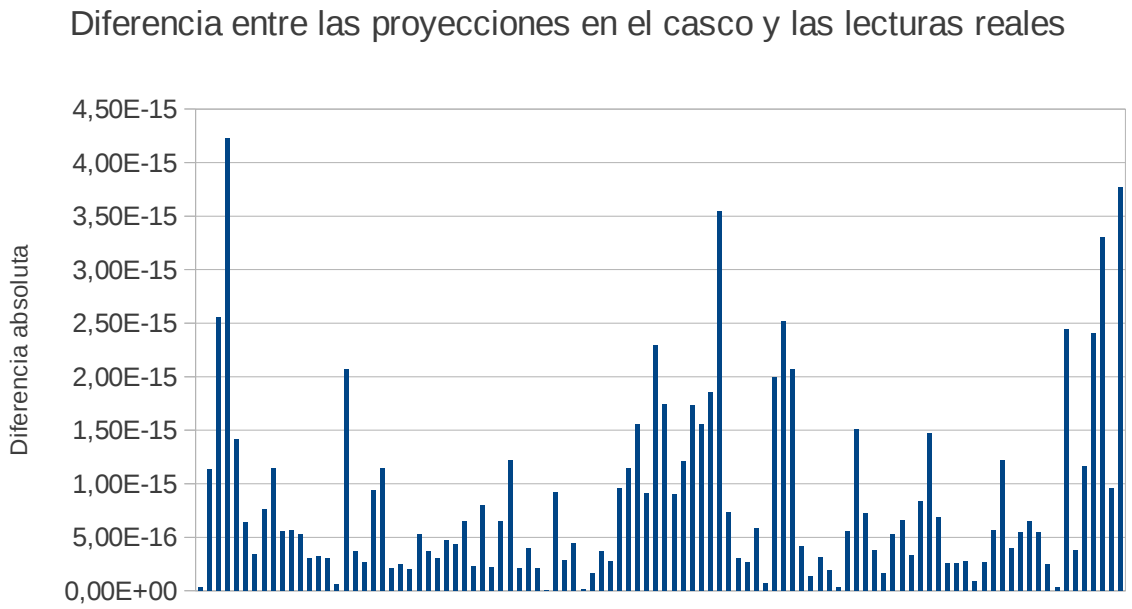

Figura 32: Diferencia absoluta entre las mediciones del casco y la proyección de la solución mediante la matriz leadfield para la solución representada en la Figura 31.

La Figura 31 muestra las diferencias en valor absoluto entre los valores de la solución proporcionada por el algoritmo *beamforming* y la proporcionada por el algoritmo Solis-Wets habiendo empleado el último una solución constante $2.29*10^{-7}$ como solución inicial. Este valor es el valor medio de una solución obtenida por medio de la aplicación del algoritmo Solis-Wets con una solución inicial 0.

La media de los valores de la figura 31 es 0.000188 y la desviación típica 0.000238

El error de la solución proporcionada por el algoritmo Solis-Wets es $1.47*10^{-28}$

La figura 32 muestra la diferencia entre las lecturas reales del casco y las que se hubieran medido si la solución proporcionada por el algoritmo Solis-Wets fuera exacta.

La media de estos valores es $8.45*10^{-16}$, y la desviación típica, $8.59*10^{-16}$

7.5 Solución inicial a $1.67*10^{-5}$ (Promedio de una solución comenzada en beamforming)

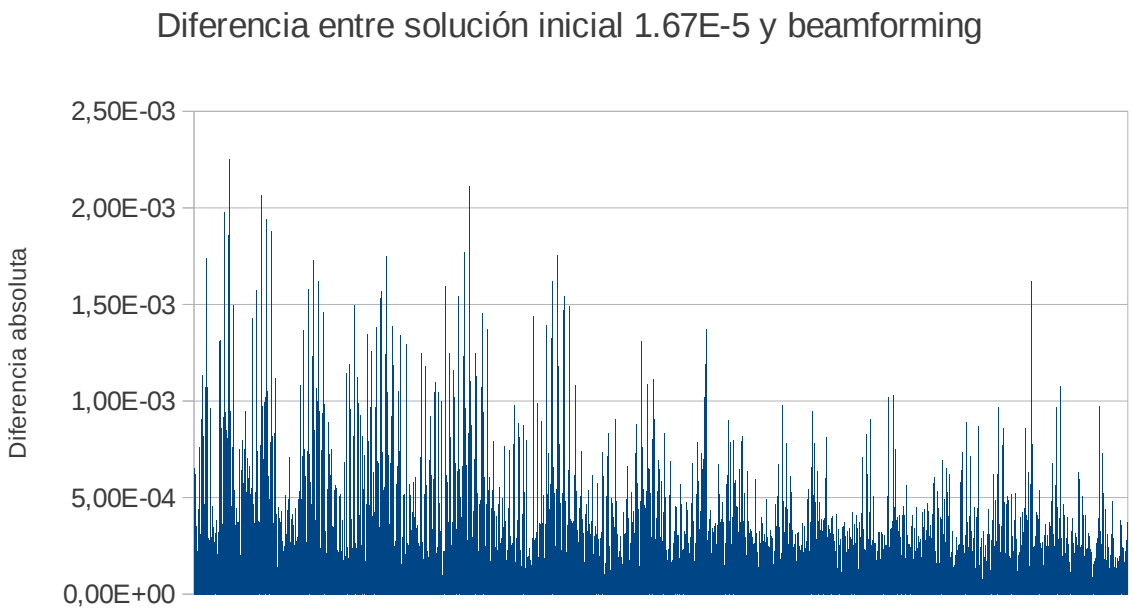

Figura 33: Diferencia absoluta entre los valores de la solución proporcionada por el algoritmo beamforming y los proporcionados por el algoritmo Solis-Wets comenzando una solución formada completamente por 1.67E-7 tras 10^6 soluciones generadas.

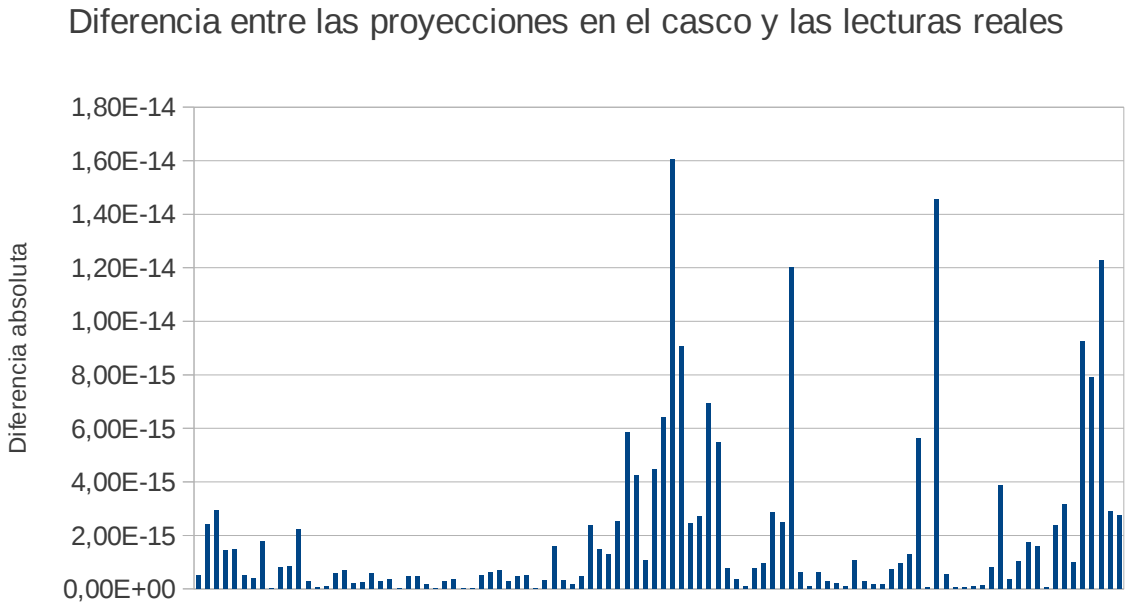

Figura 34: Diferencia absoluta entre las mediciones del casco y la proyección de la solución mediante la matriz leadfield para la solución representada en la Figura 33.

La Figura 33 muestra las diferencias en valor absoluto entre los valores de la solución proporcionada por el algoritmo *beamforming* y la solución proporcionada por el algoritmo Solis-Wets habiendo empleado el último como solución inicial una solución constante $1.67*10^{-5}$. Este valor es el valor medio de una solución obtenida aplicando el algoritmo Solis-Wets a la solución del *beamforming*.

La media de los valores de la figura 31 es 0.000225 y la desviación típica 0.000240

El error de la solución proporcionada por el algoritmo Solis-Wets es $1.36*10^{-27}$

La figura 32 muestra la diferencia entre las lecturas reales del casco y las que se hubieran medido si la solución proporcionada por el algoritmo Solis-Wets fuera exacta.

La media de estos valores es $1.94*10^{-15}$, y la desviación típica, $3.11*10^{-15}$

7.6 Conclusiones

Como se puede apreciar en las figuras 25 a 34, los mejores resultados, es decir, los que generan la solución con menor error, son los generados por la herramienta propuesta en este trabajo, utilizando como solución inicial una solución constante.

Por otro lado, aunque parezca contraintuitivo, la solución obtenida de la aplicación secuencial de ambos algoritmos, aunque mejor que los resultados obtenidos al empezar en una solución aleatoria, no es mejor que la obtenida al empezar en una solución constante, tal como se propone desde la herramienta diseñada en mi Trabajo de Fin de Grado y propuesta en este documento.

8. Conclusiones generales

De este trabajo se pueden extraer conclusiones de varias partes, de la optimización me gustaría destacar:

- La vectorización es una técnica que permite optimizar sin la necesidad de invertir demasiado esfuerzo, siempre y cuando se disponga del compilador apropiado.
- Hay pequeños cambios que pueden aportar mejoras sustanciales, como la reducción de divisiones mostrada en el apartado 6.8.
- Tener una gran cantidad de procesadores disponibles no siempre equivale a poder tener un *speedup* correspondiente al paralelizar una aplicación secuencial.
- Las bibliotecas externas son muy útiles para ahorrar trabajo, pero es muy complicado controlar cómo trabajan a bajo nivel. Esto hace que en algunos casos sea necesario prescindir de las bibliotecas durante la optimización de un código.
- Los límites de resolución de coma flotante son alcanzables.

De la parte de aplicación del algoritmo:

- Los mejores resultados, es decir, los que generan la solución con menor error, son los generados por la herramienta propuesta en este trabajo, utilizando como solución inicial una solución constante.
- La solución obtenida de la aplicación secuencial de ambos algoritmos, aunque mejor que los resultados obtenidos al empezar en una solución aleatoria, no es mejor que la obtenida al utilizar la herramienta con una solución constante como solución inicial.
- Las solución inicial aleatoria no es un buen punto de partida en comparación con el resto.

Y como conclusión global:

- Un gestor de versiones es una herramienta imprescindible.

9. Líneas de trabajo futuro

Esta herramienta pretende proporcionar una visión más precisa de los mecanismos de conductividad en el cerebro, con lo cual puede utilizarse para mejorar la investigación en campos como:

- Neurogénesis y migración neuronal.
- Exploración de enfermedades relacionadas con el sistema nervioso profundo o el sistema límbico.
- Diseño medicamentos específicos.
- Relación entre la actividad cerebral y el comportamiento externo del individuo.
- Distribución y concentración de neurotransmisores.
- Modificaciones estructurales que ocurren con enfermedades neurodegenerativas como el párkinson, alzheimer, etc.

10. Bibliografía

Problema directo de la búsqueda de fuentes:

Guido Nolte (2003). The magnetic lead field theorem in the quasi-static approximation and its use for magnetoencephalography forward calculation in realistic volume conductors. In Physics in Medicine and Biology Vol 48 No 22 (Oct., 2003)

Algoritmo Solis-Wets

Francisco J. Solis and Roger J-B. Wets . (1981). Minimization by Random Search Techniques. In Mathematics of Operations Research, Vol. 6, No. 1 (Feb., 1981), pp. 19-30

COLINY. Coliny User Manual Version 1.0 [en línea]. [Fecha de consulta: 25 febrero 2013]. Disponible en: https://software.sandia.gov/Acro/releases/votd/acro/packages/coliny/doc/uguide/html/soliswets-doc.html

Manual valgrind

VALGRIND Developers. Valgrind [en línea]. [Fecha de consulta: 09 abril 2013]. Disponible en: http://valgrind.org/docs/manual/manual.html

Algoritmo beamforming: toolbox fieldtrip

FIELDTRIP. documentation [en línea]. Actualizada: 07 febrero 2012. [Fecha de consulta: 14 mayo 2013]. Disponible en: http://fieldtrip.fcdonders.nl/documentation

Bibloteca uBLAS

WALTER Joerg, KOCH Mathias, WINKLER Gunter, STEVENS Michael. uBLAS operations overview- 1.53.0 [en línea]. [Fecha de consulta: 15 abril 2013] disponible en: http://www.boost.org/doc/libs/1_53_0/libs/numeric/ublas/doc/operations_overview.htm

Biblioteca Eigen

BENOÎT Jacob. Eigen. Actualizada: 31 mayo 2013 [Fecha de consulta: 31 mayo 2013] Disponible en: http://eigen.tuxfamily.org/index.php?title=Main_Page

Biblioteca mkl:

INTEL. Intel® Math Kernel Library (Intel® MKL) 11.0. [en línea] [Fecha de consulta: 29 abril 2013]. Disponible en: http://software.intel.com/en-us/intel-mkl

INTEL. Getting Started Tutorial: Using the Intel® Math Kernel Library for Matrix Multiplication [en línea]. [Fecha de consulta: 30 abril 2013]. Disponible en: http://software.intel.com/sites/products/documentation/doclib/mkl_sa/11/tutorials/mkl_mmx_c/tutorial_mkl_mmx_c.pdf

INTEL. Intel® Math Kernel Library Link Line Advisor [en línea]. [Fecha de consulta: 29 abril 2013]. Disponible en: http://software.intel.com/en-us/articles/intel-mkl-link-line-advisor/

INTEL. New matrix vector product BLAS routines [en línea]. [Fecha de consulta: 30 abril 2013]. Disponible en: http://software.intel.com/en-us/articles/new-matrix-vector-product-blas-routines

OpenMP

OPENMP.ORG. OpenMP3.1-CCard.pdf [en línea]. [Fecha de consulta: 06 mayo 2013]. Disponible en: http://openmp.org/mp-documents/OpenMP3.1-CCard.pdf

Vectorización

INTEL. A Guide to Vectorization with Intel® C++ Compilers [en línea]. [Fecha de consulta: 16 mayo 2013] Disponible en: http://download-software.intel.com/sites/default/files/m/d/4/1/d/8/CompilerAutovectorizationGuide.pdf

VLADIMIROV Andrey. Auto-Vectorization with the Intel Compilers: is Your Code Ready for Sandy Bridge and Knights Corner? [en línea]. Actualizada: 12 marzo 2012. [Fecha consulta: 17 mayo 2013] Disponible en: http://research.colfaxinternational.com/file.axd?file=2012%2F3%2FColfax_Sandy_Bridge_AVX.pdf

Método de Box-Muller

WOLFRAM Research. Box-Muller Transformation [en línea]. [Fecha de consulta: 11 marzo 2013] Disponible en: http://mathworld.wolfram.com/Box-MullerTransformation.html

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Fri Feb 14 20:19:45 CET 2014
	Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
	Numero de Serie	630
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)